# Hybrid Designs for Caches and Cores

by

Faissal Sleiman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2015

Doctoral Committee:

Associate Professor Thomas F. Wenisch, Chair
Professor Scott Mahlke
Assistant Professor Jason Mars
Associate Professor David Wentzloff

To my wife, Elizabeth,
who is my companion in everything.

To my parents, Mohamad and Rasha,
who have supported me all my life.

And to my sisters, Farah and Haya,
who make me a better person.

# ACKNOWLEDGEMENTS

In retrospect, I may have begun my Ph.D. journey without much idea of what I was undertaking. For the support and guidance that led me through, I would like to extend my sincerest gratitude to my committee chair and dissertation advisor, Thomas Wenisch. I would also like to thank the rest of the committee members and collaborators that helped supervise my work: Scott Mahlke, Jason Mars, David Wentzloff, Ronald Dreslinski and Reetuparna Das. Further, many administrators always welcomed my sudden requests; thank you Lauri Johnson-Rafalski, Stephen Reger, Denise Duprie, Dawn Freysinger and everyone behind the scenes. Thank you to my labmates, for the impromptu whiteboard idea-bouncing, the walks to get lunch and coffee, and putting up with my incompetence at StarCraft; including Steven Pelley, Aasheesh Kolli, Prateek Tandon, Jeffrey Rosen, Neha Agarwal, Richard Sampson, David Meisner and Gaurav Uttreja. I am also grateful to the labs across the hall for filling in the gaps with their range of expertise on the research, the tools, and where to get free food; particularly Andrew Lukefahr, for the incessant, early morning kitchen runs. Finally, to my tango family; Denzil Bernard, Rawan Abdel Khalek, Mani Kashanianfard, Yusuf Murgha, Roshan Joseph, Tracey Rosen, Katherine Mitroka, and Roman and Mira Krauze: thank you for keeping my Ph.D. years refreshing.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# ABSTRACT

Hybrid Designs for Caches and Cores
by
Faissal Mohamad Sleiman

Chair: Thomas F. Wenisch

Processor power constraints have come to the forefront over the last decade, heralded by the stagnation of clock frequency scaling. High-performance core and cache designs often utilize power-hungry techniques to increase parallelism. Conversely, the most energy-efficient designs opt for a serial execution to avoid unnecessary overheads. While both of these extremes constitute one-size-fits-all approaches, a judicious mix of parallel and serial execution has the potential to achieve the best of both high-performing and energy-efficient designs. This dissertation examines such hybrid designs for cores and caches. Firstly, we introduce a novel, hybrid out-of-order/in-order core microarchitecture. Instructions that are steered towards in-order execution skip register allocation, reordering and dynamic scheduling. At the same time, these instructions can interleave on an instruction-by-instruction basis with instructions that continue to benefit from these conventional out-of-order mechanisms. Secondly, this dissertation revisits a hybrid technique introduced for L1 caches, way-prediction, in the context of last-level caches that are larger, have higher associativity, and experience less locality.

# CHAPTER I

# Introduction

When considering the design space of processor components, the highest performing cache and core designs expend an exorbitant amount of energy to accomplish a task in parallel steps, often performing unnecessary operations in the process. On the other hand, energy-efficient designs tend to execute the bare essential operations required to complete a certain task, albeit in a slower, serialized fashion. Examples of high performance designs include parallel tag-data access in the cache and out-of-order execution in the core, while sequential tag-data caches and in-order cores form the other extreme. Hybrid designs attempt to achieve the best of both, parallelizing when it is most likely to be beneficial, and saving energy when parallelization is wasteful.

Our thesis spans hybrid designs for both the core and cache. The primary focus of this thesis is a novel, hybrid core microarchitecture, which allows inflight instructions to leverage out-of-order techniques when beneficial, while simultaneously steering other inflight instructions through more efficient in-order hardware. This design allows out-of-order or in-order to be selected at an instruction-by-instruction basis, while maintaining the tight scheduling requirements of high performance cores. We have also completed an investigation of hybrid caches, focusing mainly on last-level on-chip caches. With successive processor generations, these caches capture more of an application's working set. We update existing hybrid cache designs for the last-level cache, which poses new challenges.

## 1.1 Hybrid Core Design

Modern processors use a variety of microarchitectural techniques to extract application performance. Out-of-order (OOO) execution and simultaneous multithreading

[57] (SMT) are two such techniques, which seek to utilize superscalar execution resources by increasing single-threaded instruction-level parallelism and thread-level parallelism, respectively. Some designs seek to balance single-threaded performance and throughput by using both OOO and SMT hardware. However, OOO and SMT mechanisms compete to fill the same functional units using different types of parallelism. As such, prior work finds that the throughput of an in-order (INO) core approaches that of an OOO core as the number of SMT threads is increased [23]. This result indicates that expensive OOO mechanisms are not always necessary and may be an inefficient underlying microarchitecture for multi-threaded designs.

OOO hardware enables early issue of instructions that encounter false dependences, for which INO cores must stall. However, for a significant fraction of instructions, the last-arriving input operand (true dependence) arrives after all false dependences have resolved. Such instructions, which we call *in-sequence*, do not stall in INO cores and naturally issue after all elder instructions (i.e., in program order) in OOO cores. Conversely, we refer to instructions that naturally issue out of program order as *reordered*.

In-sequence instructions gain no benefit from the OOO microarchitecture structures they occupy. In fact, these instructions can be safely executed on schedule *without* allocating in OOO structures, including the reorder buffer, issue queue, load-store queue, and physical register file. We find that having more SMT threads increases the fraction of in-sequence instructions observed in a particular OOO instruction window. Additionally, in-sequence instructions interleave at fine granularity with reordered instructions. We find that groups of consecutive in-sequence or reordered instructions average 5 to 20 instructions per group. So, existing hybrid INO/OOO microarchitectures [35, 39], which switch at 1000-instruction (or higher) granularity, cannot exploit the in-sequence phenomenon without sacrificing performance on reordered instructions.

Instead, we propose a microarchitecture in Section 3.2 where in-sequence instructions occupy an energy-efficient FIFO queue, which we call the *shelf*[1], from which instructions may issue only in sequence (reordered instructions occupy a conventional, unordered issue queue). By shifting in-sequence instruction occupancy to the inexpensive shelf, capacity in OOO structures is freed for reordered instructions. Given that, as we show later, around half of instructions are in-sequence in a 4-thread SMT, we aim to double the effective instruction scheduling window simply with the allocation

[1]We borrow the naming concept for the *shelf* from the Metaflow architecture [49], as our structure is based on the principle of shelving deferred instructions.

of FIFO queues.

This thesis makes contributions in three areas. First, we report on the correlation between in-sequence instructions and the effectiveness of OOO hardware—in-sequence instructions gain no benefit from OOO mechanisms. Second, we leverage that insight to design a microarchitecture that integrates a shelf into an SMT-enabled out-of-order core. To our knowledge, this is the first design that enables a modern dynamically scheduled instruction window, with instruction reordering and register renaming, to contain statically scheduled, unreordered instructions, which reuse the same registers, chosen at instruction granularity. Finally, we describe a simple hardware steering mechanism that determines whether instructions must be steered to the shelf based on whether they are predicted to be in-sequence or reordered in the future schedule.

Our design achieves an 8.3% energy-delay improvement over the best baseline 4-thread OOO core, because it is more efficient to extend an OOO instruction window with a shelf than it is to expand conventional OOO hardware structures. We explore the limits of our technique across a number of OOO sizes and SMT threads. For instance, we show that a shelf is less amenable to single-threaded cores because they encounter fewer in-sequence instructions for the same core size.

## 1.2 Hybrid Cache Design

Semiconductor device scaling continues to enable processor designs with ever-larger caches. As larger working sets are captured within the chip, the importance of intra-chip access latency has grown [15]. Server applications are particularly sensitive to last-level cache (LLC) latency because their multi-megabyte instruction footprints overwhelm primary instruction caches, exposing LLC latency on the fetch critical path [16]. While tag and data accesses in large, highly-associative LLCs are often serialized to save energy [50], parallel tag-data access (for reads) can reduce overall access latency by 30% albeit at a 1.47x cost in per-access energy.

To bridge the performance and energy gaps between these two extremes, we consider *way prediction*, where only a subset of data ways are accessed in parallel with the tags. Way prediction has been studied extensively for L1 caches [9, 24, 31, 38, 50, 62]. In this context, it has relied primarily on one of two phenomena: temporal locality (i.e., predict the most-recently-used way) or instruction-correlated locality (i.e., use a PC-indexed prediction table). However, way prediction is fundamentally harder in LLCs because associativity is greater, temporal locality is filtered by the L1

caches, accesses from multiple cores interleave, and instruction addresses are typically not available.

Alternatively, researchers have advocated *partial tag comparison* to rule out cache ways that surely do not contain the data [11, 28, 42, 63]. These designs compare a few low order bits of the incoming tag to those stored in each way, and abort accesses for any mismatches. The most recent design [63] targets small (8-32KB), low-associativity (4-way) L1 caches, which allows it to hide a fully-associative 4-bit partial tag comparison under the data array decoder delay. This design avoids any impact on the cache access critical path, while achieving good energy efficiency. The comparison is implemented with static logic as a content-addressable memory (CAM)—a design facilitated by the small L1 size.

We follow a similar approach, however our target LLC context leads us to a different solution. (1) We demonstrate the need for a wider partial tag comparison of 6-8 bits in order to achieve highly accurate way prediction (over 90% accuracy) at the LLC. (2) This wider comparison leads us to implement the CAM with dynamic logic to minimize latency, and we assess the impact on access latency by performing a circuit-level critical path analysis of the CAM versus decoder delays. (3) Despite a wider partial tag comparison, some accesses still result in partial tag matches in more than one way, which makes the way prediction inconclusive. To tightly limit energy per access, we describe an architectural feature we call the *inhibit bit* to predictively activate the most likely way under such *partial tag collisions*.

Integrating these components, we propose *Embedded Way Prediction*, an architecture and circuit design for effective way prediction in server-class LLCs. We show that embedded way prediction achieves the full potential performance improvement of parallel lookup, improving scientific and server application performance by up to 15.4% (6.6% average) at an energy-per-instruction overhead of 11%, as compared to a 17.5% overhead for conventional parallel lookup (averages are geometric means).

# CHAPTER II

# In-sequence Instructions

*Simultaneous multithreading out-of-order cores waste a significant portion of structural out-of-order core resources on instructions that do not need them. These resources eliminate false ordering dependences. However, in such cores, nearly half of instructions dynamically issue in program order after all false dependences have resolved. These* in-sequence *instructions interleave with other* reordered *instructions at a fine granularity within the instruction window.*

## 2.1 Overview

We identify three types of ordering dependences that cause simple INO cores to stall, but do not stall OOO cores. These are data, speculation and structural dependences. *Data dependences* govern the order in which register reads and writes must be performed. These comprise the well-known Read-After-Write (RAW) or true dependence, as well as the Write-After-Write (WAW) and Write-After-Read (WAR) false dependences. *Speculation dependences* involve speculative execution of instructions, including processor speculation on control flow, such as after a branch or excepting instruction, or on values, such as those returned by loads executed early in memory order. *Structural dependences* represent resource constraints. These prevent instructions from proceeding to the next stage, as in pipeline stalls in an INO core, or a dispatch stall in an OOO core when the issue queue or other structure is full.

We consider an instruction to be *reordered* if it issues to functional units before all three types of dependences are resolved, otherwise the instruction is *in-sequence*. Consider a simple INO core that stalls at the issue stage until all dependences resolve, such that all instructions are in-sequence. The simple INO core stalls for true and false data dependences by issuing instructions in program order, which takes care of WAR hazards, and with the use of a register ready bit-vector, which can detect

RAW and WAW dependences. Speculation dependences can be handled at the issue stage using Smith and Pleszkun's result shift register [53]. Structural dependences are honored automatically by the FIFO nature of the INO core. These mechanisms conservatively stall each instruction until it is guaranteed not to violate any ordering dependences. As such, all instructions in the simple INO core are in-sequence.

In-sequence instructions in OOO cores are those instructions that issue according to the same schedule (relative to other instructions) as they would have issued in an INO core. We show that in-sequence instructions waste OOO resources using a dependence graph model of the false ordering dependences. These instructions are particularly abundant in SMT-enabled processors. Additionally, our observations reveal that reordered and in-sequence instructions interleave at fine-granularity in OOO cores. We demonstrate in this thesis a practical OOO microarchitecture where in-sequence instructions can be selected on an instruction-by-instruction basis and skip allocation in OOO structures, *while still executing in the same schedule as a conventional OOO core.*

## 2.2 Modeling Ordering Dependencies

Fields, Rubin, and Bodik [17] develop a dynamic dependence-graph model for OOO execution to analyze its critical path. We extend their model to include the false dependences that arise in in-order cores. The extended model allows us to precisely define in-sequence and reordered instructions and understand why OOO mechanisms are unnecessary for in-sequence instructions. In later sections, we base the design of our microarchitecture on the analysis of this extended model to demonstrate why our hardware still maintains correctness.

The Fields model abstracts execution into three operations (dispatch, execute, and commit) represented by nodes, and dependencies among operations, represented by edges weighted by latencies. We show a small example of this model in Figure 2.1 (left). To additionally reason about false dependencies, we expand their execute node into three: issue (I), execute (X), and writeback (W). We depict these additional nodes in Figure 2.1 (right).

In our extended model, I corresponds to the operation of scheduling instructions to functional units, X corresponds to the computation of instruction outputs, and W corresponds to overwriting architectural state with the results of an instruction. At the I node, an instruction waits for its operands (true dependences), for dispatch (the

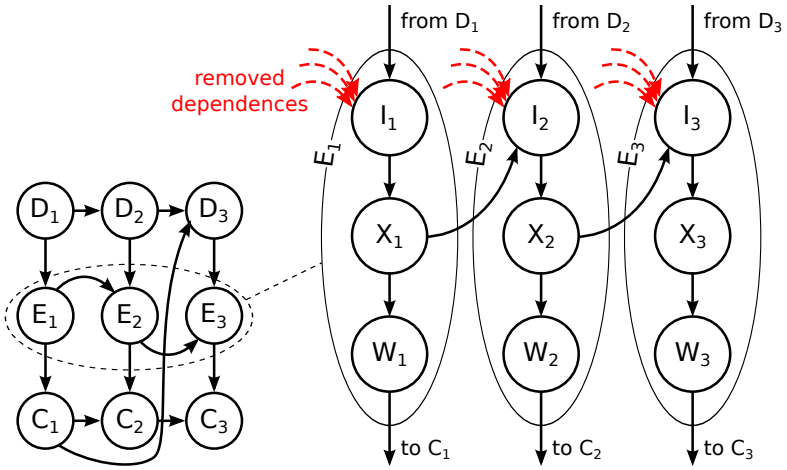



Figure 2.1: Dependence chain model for issue stage.

previous operation)[1]. A node (or operation) triggers after the last-arriving edge (or dependence). Once an instruction has dispatched and its operands are available, the I node triggers, and the instruction *issues* to the functional units. The X node then triggers, incurring the latency of the functional unit in the X-W edge which represents execution latency. Operand dependences are signified by X-I edges. Finally, the instruction writes back in the W node.

An instruction is *reordered* if the last-arriving dependence at any node is a false dependence. Such instructions benefit from OOO hardware; in an INO processor, the instruction must stall to await the false dependence or execution may become incorrect (e.g., architectural state required for correct recovery of a misspeculation may be lost). Conversely, if none of the false dependences are last arriving, an instruction is *in-sequence*. We will demonstrate that in-sequence instructions need not allocate in OOO structures to safely execute as scheduled by the OOO core. Instead, we allocate in-sequence instructions to an energy-efficient FIFO queue called the shelf, and conservatively enforce false dependences with practical detection mechanisms. Although we enforce these dependences, since they are not last-arriving, execution of in-sequence instructions is not delayed. (Note that, as in-sequence instructions are heuristically/predictively identified, our design must still check and enforce false dependencies for the shelf to guard against cases where an instruction is incorrectly steered).

To make our analysis of false dependencies concrete, we discuss them with respect to a simplified OOO core depicted in Figure 2.2. The figure depicts the operations

[1]Note that Fields et al. represent other dependences like functional unit availability in their model with additional latency in outgoing edges.

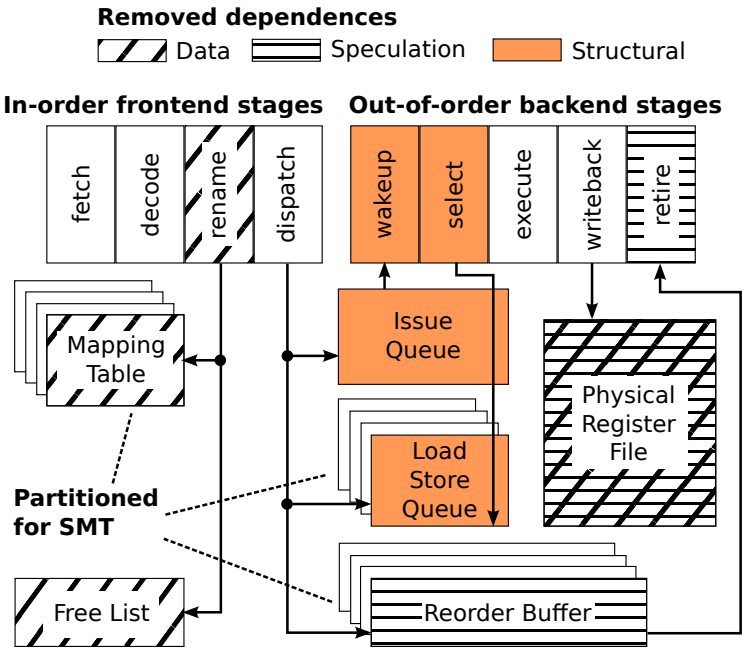
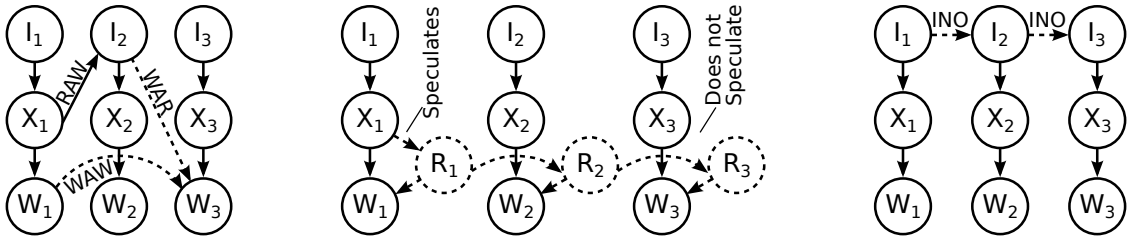



Figure 2.2: Baseline microarchitecture with SMT partitioning and OOO dependence removal.

that occur in the in-order frontend and out-of-order backend and the major structures of the core. (Note that the operations we depict do not correspond precisely to boundaries of pipeline stages in a typical core; for example, wakeup is often performed as part of other stages). The operations and structures are marked with particular false dependences they affect. Instructions proceed in program order through the frontend until they are dispatched to the backend OOO structures. For each false dependence, we show how it manifests as a dependence edge in our model, we discuss how OOO structures alleviate it, and demonstrate that an in-sequence instruction can safely issue without allocating in OOO structures.

### 2.2.1 Data

Data dependences govern the order in which register reads and writes must be performed. These comprise the well-known Read-After-Write (RAW or true dependences), as well as the false dependences: Write-After-Write (WAW) and Write-After-Read (WAR). For WAW and WAR dependences, each write must wait for preceding reads and writes to the same register to complete before it can change the value. These constitute edges to the writeback operation of the dependent instruction, sourced at the preceding write for WAW, and at the issue operation for any readers intervening between the two writers (WARs). Figure 2.3a illustrates this.




(a) Data (b) Speculation (c) Structural

Figure 2.3: Removed dependences

OOO cores employ register renaming to remove WAW and WAR dependences by allocating and tracking a new register for each write. This renaming mechanism ensures there are no preceding reads and writes to the register. The rename operation in our OOO core model carries out register renaming. We employ the mapping table (MT) to track the most recent alias of a register, and the free list (FL) to find an available register in the physical register file (PRF). The PRF provides enough register entries for all the instructions we would like to simultaneously rename. An instruction that issues in-sequence can overwrite the previous value for the register and need not rename its destination; once WAW and WAR dependences have passed, there is no need for the previous value. (Note, misspeculation recovery may still require the value, which we address next.)

### 2.2.2 Speculation

Modern processors speculate on values, such as the next PC after a branch, or on the early execution of memory loads. Sometime during or after their execution, instructions that use speculative values verify whether these are correct or misspeculated. Recovering from a misspeculation involves squashing the instruction stream after the misspeculated instruction and re-executing. We say an instruction has *resolved* if it has no misspeculations and all preceding instructions are non-speculative. Before an instruction resolves, it is *speculative*, and cannot overwrite an old value because it may be needed for recovery. This requirement leads to a dependence edge to the writeback stage, which must carry the dependence on the current instruction's resolution as well as older instructions. To model this correctly, we add a resolution (R) node as in Figure 2.3b to each instruction to represent the verification operation wherein an instruction is confirmed not to fault or misspeculate (branch outcome resolves, divisor is non-zero, memory translation succeeds, etc.). All instructions must wait for the R node before writing back, and have an R-W edge. Speculative in-

structions also have an X-R edge to signify the verification operation. R nodes are connected in sequence with R-R edges from one instruction to the next.

OOO cores piggyback the renaming mechanism to write to a newly allocated register even as they wait their turn to resolve. The resolution order is maintained by a reorder buffer (ROB) which retires instructions in program order once they have completed execution. The old value for each register is maintained until the instruction producing the new value retires. This allows the processor to squash all instructions younger than a certain point and recover to the old values.

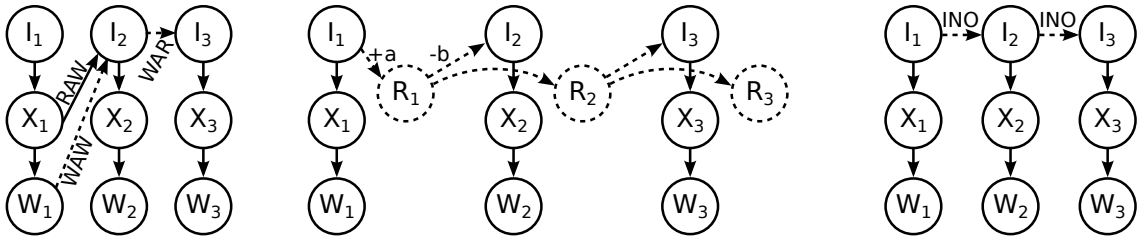
In-sequence instructions happen to write their outputs only after they are non-speculative. Therefore, they may overwrite the previous value of their destination register in-place, without any chance of roll-back. By virtue of never needing to hold onto the old value they also do not need to allocate in the ROB. All that is needed is a mechanism which prevents them from overwriting a register should they need to be squashed, which does not require the ROB.

### 2.2.3 Structural

Structural dependences represent resource constraints. Particularly in INO cores, instructions must issue in program order as they cannot leap-frog. OOO cores remove these issue-to-issue dependences (I-I edges in Figure 2.3c) by allocating associative structures like the issue queue (IQ) and load-store queue (LSQ). These structures involve broadcast/wakeup mechanisms to notify all entries of updates to their dependences, as well as mechanisms to select instructions from any entry to execute. The broadcast-based wakeup and associative selection remove the issue-to-issue dependence from one instruction to the next. In-sequence instructions can avoid costly associative structures and opt for FIFO structures to issue in program order.

## 2.3 Designing for In-sequence Instructions

Data, speculation and structural dependences are not removed in INO cores, which must handle these by stalling to avoid correctness hazards. To facilitate dependence tracking, INO cores issue instructions in program order and determine stall conditions at the issue stage. In-sequence instructions are instructions that would not stall under INO hardware. Our design aims to facilitate their execution using INO-like mechanisms, avoiding the more energy-intensive OOO hardware while ensuring correct, timely execution.




(a) Data (b) Speculation (c) Structural

Figure 2.4: Converted dependences

Our dependence-graph models show that removed dependences do not all arrive at the issue node. However, in our hardware design, we seek to detect and stall for all dependences at issue. Hence, in Figure 2.4, we transform all dependences to arrive at the issue node while conservatively ensuring that in-sequence instructions may issue no earlier in the transformed model than in the original. Structural dependences already arrive at the issue node. WAW and WAR edges can simply be moved to point to the issue node, enforcing the original constraint via transitivity. We note that WAR edges are now subsumed by structural dependence edges: an instruction that issues in program order cannot violate a WAR dependence because older instructions have already issued and read their operands.

To transform speculation dependences, we ensure that an instructions stalls at the issue node until it is guaranteed to writeback after all elder speculations resolve. We reposition the R-nodes between consecutive issue operations and calculate an edge weight that ensures enough delay that the resolution-to-writeback ordering constraint is enforced (i.e., the second instruction will stall if the new R-I edge is last-arriving). The original X-R edge becomes an I-R edge and is annotated with the resolution latency of the elder instruction (+a). The R-W edge becomes an R-I edge to the younger instruction and is annotated with the younger instruction's minimum execution latency (-b). The difference in latency (a-b) is the number of stall cycles needed to ensure correctness. R-nodes still carry the R-R edges as before. Together, these transformations replace the transitive $X_1 \rightarrow R_1 \rightarrow R_2 \rightarrow W_2$ dependence chain with a $I_1 \rightarrow R_1 \rightarrow I_2 \rightarrow X_2 \rightarrow W_2$ dependence chain.

Our approach requires known execution and resolution latencies for all instructions. Although many instructions have variable execution latency, the time to detect potential faults (e.g., a load instruction misses in a TLB, a floating point division by zero) is typically short and fixed. Prior designs (e.g., [53]) have made the same assumption.

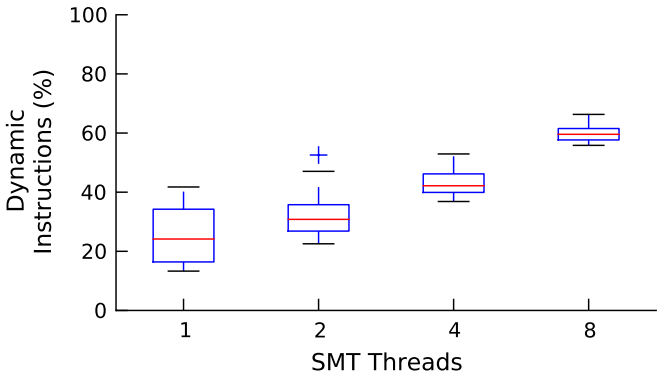



Figure 2.5: Fraction of instructions wasting OOO resources.

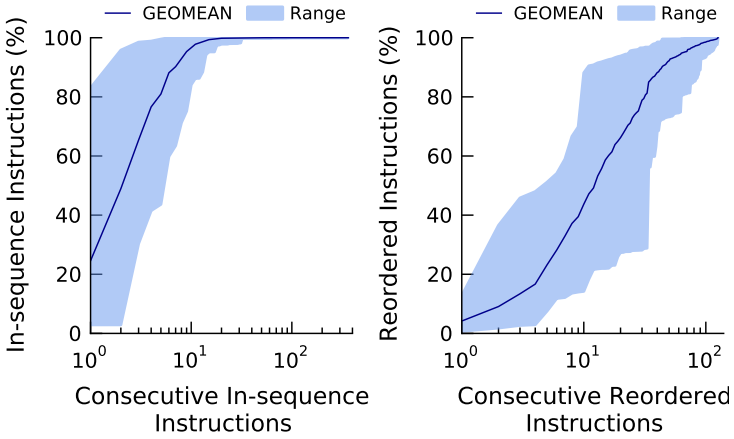



Figure 2.6: Weighted cumulative distribution of consecutive in-sequence and reordered instruction series lengths.

## 2.4 Opportunity

We have shown that in-sequence instructions need not allocate in OOO structures. By removing these instructions, we reduce pressure on these structures and allow the designer to shrink them, reducing power while maintaining performance. A significant fraction of instructions in OOO cores execute in-sequence, and this number increases with number of SMT threads in a core. Figure 2.5 illustrates the extent of this effect; as the number of threads in a 128-entry OOO instruction window is increased, the fraction of in-sequence instructions more than doubles to more than 50% on average. As such, we target the performance of the 128-entry instruction window, using a 64-entry one with minimal additional hardware for in-sequence instructions.

We show that in-sequence and reordered instructions interleave at fine granularity. Figure 2.6 depicts the cumulative distribution of consecutive in-sequence and reordered series lengths as seen in an OOO core, weighted by the number of instruc-

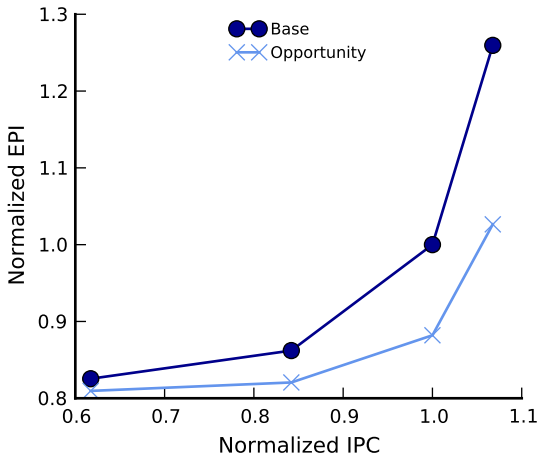



Figure 2.7: Energy-efficiency potential of hybrid design on 4-thread SMT.

tions in the series (the series length). The plot shows the geometric mean across benchmarks, as well as their range of behavior, for single-threaded benchmarks. We find that 99% of in-sequence instructions occur in series with 30 instructions or fewer, while a series of reordered instructions is bound by the ROB size (128 entries in this case). SMT workload mixes with 2, 4 and 8 threads generally produce similar distributions.

Figure 2.7 shows a back-of-the-envelope estimate of opportunity for energy savings if one were to down-size OOO structures by the average fraction of in-sequence instructions while maintaining the same performance as the original OOO design. The in-sequence instructions are instead assumed to occupy a FIFO buffer equal in size to the portion of the ROB that was removed. Replacing out-of-order mechanisms with a shelf provides substantial opportunity to improve energy efficiency.

Whereas this simple estimate presupposes that individual OOO structures can be scaled arbitrarily, in fact, many structures naturally lend themselves to implementations at power-of-two sizes. Hence, in our experimental evaluation, we instead compare the energy-delay of practically sized OOO and OOO+Shelf microarchitectures, demonstrating the improvement in pareto-optimal design points enabled by our approach.

# CHAPTER III

# Hybrid In-Order/Out-Of-Order Core Microarchitecture

*We develop a hybrid out-of-order/in-order microarchitecture, which can dispatch instructions to efficient in-order scheduling mechanisms—using a FIFO issue queue called the* shelf*—on an instruction-by-instruction basis. Instructions dispatched to the shelf do not allocate out-of-order core resources in the reorder buffer, issue queue, physical registers, or load-store queues.*

## 3.1 Design Overview

OOO designs extend the OOO instruction window by provisioning reorder buffer (ROB), issue queue (IQ), load-store queue (LSQ), and physical register file (PRF) entries. These structures are provisioned in a balanced fashion so that no one structure is a dominant bottleneck. Larger OOO cores thus increase the instruction window by increasing all structures. Our main observation is that in-sequence instructions do not need these costly structures to execute correctly on schedule. Nevertheless, the instructions must still be buffered so as to extend the OOO instruction window. We provide this buffering via a per-thread in-order issue queue, which we call a *shelf*.

A shelf is a FIFO buffer that holds instructions in between the dispatch and issue stages much like the (fully associative) IQ. It serves to unblock the dispatch stage to allow reordered instructions to proceed past stalled in-sequence instructions. Shelf instructions are not allocated a new PRF or ROB entry. As such, shelf issue logic must detect and handle false dependences by stalling. Ideally, instructions steered to the Shelf are in-sequence instructions, which do not incur additional stalls for false dependences. We discuss steering instructions to the shelf or IQ in Chapter IV.

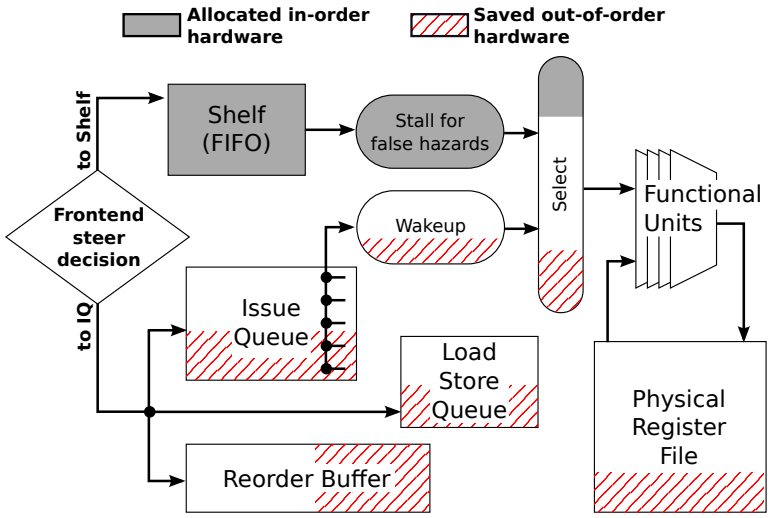



Figure 3.1: Design overview with FIFO shelf.

Figure 3.1 illustrates the shelf within our generic OOO pipeline. It depicts incoming instructions from the dispatch stage as steered to the shelf or to the IQ. The steering mechanism may interleave shelf and IQ instructions on an instruction-by-instruction basis. Once instructions have dispatched to the shelf and IQ, the microarchitecture must correctly and quickly resolve true and false dependences across the two queues to prevent unnecessary stalls. Prior hybrid INO/OOO microarchitectures [35, 39] cannot exploit such fine-grain interleaving, while our design can. As the average series length of in-sequence or reordered is on the order of 10 instructions, the instruction window will simultaneously contain multiple series that interdepend. The next section focuses on the details of our mechanism.

## 3.2 A Hybrid Instruction Window

The FIFO shelf is designed to avoid costly associative operations like the tag comparison in the IQ as well as store-to-load forwarding and memory order violation detection in the LSQ. We implement the shelf as a circular buffer with head and tail pointers, much like the ROB. All shelf instructions will block behind a stalled head instruction even if they are ready to issue. Each instruction at the head of the shelf will check for false and true dependences before issuing to the functional units.

Ideally, the shelf would require only mechanisms like the simple INO core to guarantee that instructions issue in-sequence. However, in-order issue is complicated by the dynicamically-scheduled OOO instruction window. To detect false dependences inexpensively, we maintain the invariant that instructions issue from the shelf in pro-

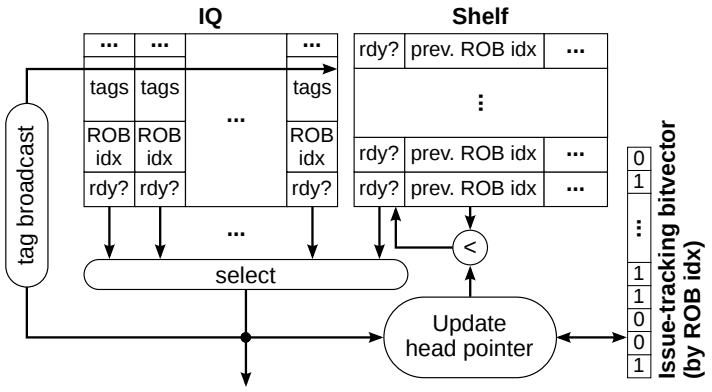



Figure 3.2: In-order issue of shelf instructions.

gram order. The main implication of INO shelf issue is that shelf instructions must issue after preceding IQ instructions, which we describe in Section 3.2.1. We also modify the stalling mechanisms for speculation (Section 3.2.2) and data dependences (Section 3.2.3). Finally, we discuss memory ordering in Section 3.2.4.

### 3.2.1 Issuing from the Shelf in Program Order

By virtue of the shelf being a FIFO buffer, its instructions are already ordered with respect to each other. So, an instruction at the head of the shelf need only stall for unissued instructions from the immediately preceding series of IQ instructions (earlier series of IQ instructions must have already issued for the shelf instruction to reach the head). For this reason, we designate that a new *run* of instructions starts when an IQ instruction is steered immediately following a shelf instruction from the same thread. One run consists of a series of IQ instructions followed by a series of Shelf instructions. An instruction at the head of the shelf that issues after all IQ instructions in the same run is guaranteed to issue in program order.

Since IQ instructions are dynamically scheduled, the first instruction to dispatch is not necessarily the first one to issue. Additionally, consecutive instructions are generally not allocated adjacent entries in an IQ. To track the issue order of IQ instructions, we allocate a per-thread issue-tracking bitvector with one bit per ROB entry, which represents whether the corresponding instruction has yet to issue (see Figure 3.2). The bit corresponding to an instruction is cleared upon dispatch, and set upon issue. A head pointer is maintained to track the oldest unissued IQ instruction, similarly to how the ROB tracks the oldest instruction that has not retired. To be eligible for issue, a shelf instruction must ensure that the head pointer has moved past the last IQ instruction in its run. So, as an instruction is dispatched to the shelf,

it records the ROB index of the last preceding IQ instruction (i.e., the tail pointer of the ROB/issue-tracking bitvector for its thread). Once the head pointer advances past this index, the shelf head is the eldest unissued instruction and can proceed to issue in program order.

#### 3.2.1.1 Critical Path Considerations.

In a superscalar machine, we may desire to issue a shelf instruction in the same cycle as the last older IQ instruction. We consider the circuit-level critical path challenges associated with same-cycle issue. An issue cycle consists of selecting a number of ready instructions, followed by waking up their dependents to mark them ready for the next cycle. To determine if the head of the shelf is eligible for issue, the issue-tracking bitvector must be updated to reflect the elder IQ instructions selected for issue this cycle. Aggressive, same-cycle issue of an IQ instruction and subsequent shelf instructions requires this combinational logic to be placed on the critical path of wakeup and select, which are typically already among the longest paths in OOO processors.

Instead, we advocate a conservative design as depicted in Figure 3.2, which does not bypass issue-tracking bitvector updates, removing these updates from the wakeup-select critical path. Although shelf instructions cannot issue in the same cycle as preceding IQ instructions, we confirm that this design choice sacrifices minimal performance later in Section 5.2.

As the shelf extends the instruction window, it effectively competes against larger OOO cores with longer critical paths. We evaluate various OOO sizes under the same clock frequency to isolate microarchitectural effects; nevertheless, we assume that small critical path overheads induced by our shelf design compare favorably to the critical paths in larger, slower designs.

### 3.2.2 Handling Speculation

Instructions are considered *committed*, or no longer speculative, once they have resolved their speculation and all older instructions are committed. To guarantee correct execution, instructions must not overwrite or recycle old state until they are committed. The ROB is the conventional OOO structure that maintains the commit order by retiring old state in program order after each instruction writes back (which guarantees that it has resolved speculation). Since shelf instructions do not allocate in the ROB, they must be delayed at issue until they are guaranteed to write back

in-sequence. This constraint allows shelf instructions to reuse the physical register previously allocated for their destination register index. We discuss the shelf delay mechanisms below, then discuss how to squash shelf instructions and prevent them from writing back on a misspeculation. Finally we consider coordinating the ROB retire order with shelf instructions.

#### 3.2.2.1 Delaying Shelf Instructions for Speculation.

We first describe the simplest method to delay shelf instruction writeback correctly. This method is based on the result shift register proposed by Smith and Pleszkun in the context of in-order cores with varying but deterministic instruction execution latencies [53]. We introduce a *speculation shift register* (SSR) per thread, which tracks the maximum remaining resolution cycles for any in-flight instruction. As each speculative instruction issues, it sets the SSR to the maximum of its resolution delay and the current SSR value. Since shelf instructions issue in program order, when the instruction at the head of the shelf is eligible for issue, the SSR will have been updated by all older instructions. A shelf instruction can only issue once its minimum execution delay compares less than or equal to the value in the SSR. Any earlier and it becomes unsafe to issue the shelf head (i.e., it could overwrite the value in its destination register, which is later needed for recovery).

Although the mechanism we have described thus far maintains precise state, it can unnecessarily delay shelf instructions due to speculative execution of younger reordered instructions; such younger instructions may issue early, merging their resolution time into the SSR. In pathological cases, the shelf head may be the eldest incomplete instruction and yet stall indefinitely, until the issue of all younger instructions becomes blocked due to dependences on the shelf. (This pathology could not arise in Smith and Pleszkun's setting, where issue is in-order [53].) To avoid this pathology, we provision additional SSRs. We could aggressively enforce precise speculation stalls by provisioning a separate SSR for each run; however, the number of in-flight runs varies greatly over the course of execution. Moreover, to support per-run SSRs, each IQ instruction would need to track which SSR it must update.

Instead, we propose a conservative design with only two SSRs, an IQ SSR and a shelf SSR as shown in Figure 3.3. All IQ instructions update only the IQ SSR with their resolution time as they issue. Shelf instructions refer only to the shelf SSR to determine if they are safe to issue. Whenever the first shelf instruction in a particular run becomes eligible for in-order issue, the IQ SSR is first copied to the shelf SSR. At

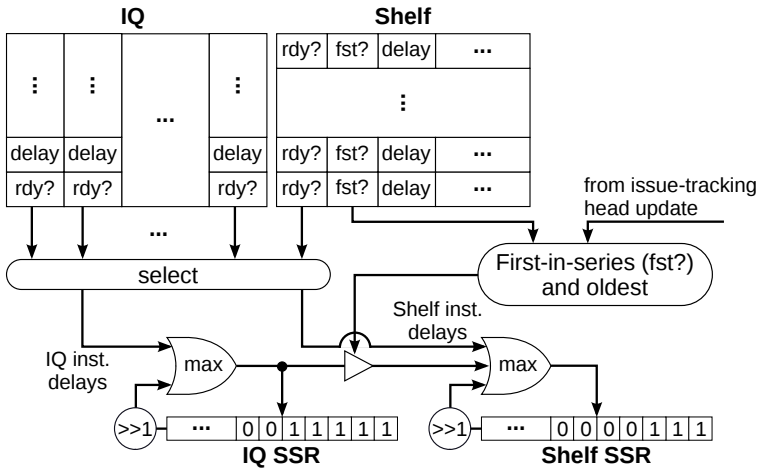



Figure 3.3: Delaying the shelf for speculation.

this moment, it is guaranteed that all elder IQ instructions have issued and updated the SSR (as the shelf head is the eldest unissued instruction).

The IQ SSR may also include the resolution delay of younger instructions that issued early, for which we (unnecessarily, but conservatively) enforce a delay. However, the starvation pathology described above is no longer possible, since no more IQ instructions will affect the shelf SSR until the shelf head issues. We evaluate the performance loss of our optimized design relative to the perfect SSR case in Section 5.2.

#### 3.2.2.2 Shelf Retirement and Squashing.

Once a shelf instruction reaches the writeback stage without being squashed, there can be no readers, writers or recoveries to the state that it will overwrite (data dependences are handled in Section 3.2.3); the issue invariants maintained by the shelf ensure writeback is safe. So, shelf instructions may retire out-of-order.

The shelf must also recover correctly from misspeculations by precisely squashing all instructions younger than the misspeculating instruction. As noted previously, when an instruction misspeculates, there can be no younger shelf instructions that have already written back. All shelf instructions that must be squashed are either unissued or still in flight in execution pipelines. These instructions (including the misspeculating instruction itself) must be prevented from writing back as they complete. There may also be elder in-flight shelf instructions, which must *not* be squashed. Hence, a misspeculating instruction must indicate precisely the index of the first shelf instruction to be squashed. This *shelf squash index* can be used to filter out younger shelf instructions as they write back.

For misspeculating shelf instructions, identifying the shelf squash index is trivial: it is the misspeculating instruction's own index. For IQ instructions, we store during dispatch the index of the first shelf instruction that is to follow: it is the index the next shelf instruction will be assigned, indicated by the shelf tail pointer.

A consequence of this recovery design is that a shelf index may not be recycled for use by another instruction until its first assignee writes back. In contrast, IQ entries may be recycled immediately once the instruction occupying them issues. A simple solution is to release shelf entries only upon writeback. However, this approach greatly increases shelf occupancy; as our goal is to squeeze the most efficiency out of as little hardware as possible, the increased occupancy is undesirable. We discuss an alternative that decouples the shelf index (which cannot be reused) from the shelf entry (which may then be used by another instruction) in Section 3.2.2.3.

Given an ordered shelf index that uniquely identifies in-flight instructions, we can now precisely writeback and retire only shelf instructions older than the shelf squash index, and rollback or squash younger shelf instructions once the retire pointer of the shelf arrives at the shelf squash index.

#### 3.2.2.3 ROB Retirement.

For IQ instructions, the ROB ensures in-order retirement with respect to other IQ instructions, thereby facilitating precise recovery from exceptions—architectural state modified by IQ instructions is updated in program order and non-speculatively. However, the ROB must also coordinate with the out-of-order retirement of shelf instructions to ensure precise state in the event of a misspeculation—ROB instructions may not retire before older shelf instructions. Absent additional checks, ROB instructions could retire before a younger shelf instruction resolves its speculations, or reads or writes the register that is being retired.

We guard against this mis-ordering by ensuring that ROB entries may not retire before elder shelf instructions retire. Shelf instruction retirement is tracked in a *shelf retire bitvector*, much like the completion bit associated with each ROB entry. Similar to the head pointer of the ROB, a shelf retire pointer advances over this bitvector, always pointing to the eldest unretired shelf index. Each ROB entry tracks the index of the next shelf instruction to follow it in program order (recall that this is the shelf squash index, discussed above, which we must already track for misspeculation recovery). Once the shelf retire pointer matches or exceeds the stored shelf index, the ROB can retire the next instruction.

Whereas this design ensures correct ordering of ROB retirement with respect to shelf instructions, it shares the downside noted in the discussion of misspeculation recovery in Section 3.2.2.2: shelf indexes may not be recycled for use by a new instruction until elder ROB entries retire. We solve this potential resource shortage by decoupling the allocation and deallocation of the (comparatively) expensive shelf entry from that of the shelf index (a virtual resource). We assume the size of the shelf is a power of two, and allow the shelf index to span a range double the shelf size. The most significant bit of the shelf index is not used when accessing shelf entries. A single shelf tail pointer is used to allocate a shelf index and the corresponding entry (i.e., ignoring the most significant bit). The shelf retire pointer continues to track the last unretired shelf index. A second reservation pointer represents the head of the shelf index space; the tail pointer may not allocate past this pointer. The shelf reservation pointer is updated only once the ROB no longer needs an entry, and prevents the shelf tail from wrapping around to allocate the reserved indices. The reservation pointer is updated by retiring ROB instructions; each sets the reservation pointer to its shelf squash index, freeing the corresponding indices for reuse.

As such, ROB instructions check that the shelf retire pointer has retired all older instructions. Shelf entries may be reused as soon as the corresponding shelf instruction issues, but the wider shelf index remains reserved until it is no longer referenced by any ROB entry.

### 3.2.3 Handling Data Hazards

To handle data hazards, shelf instructions must stall at issue until it is guaranteed that data dependences are resolved, similar to the simple INO core. Once all data dependences are resolved, in-sequence instructions from the shelf may correctly overwrite the previous value for their destination register. Our strategy, then, is to reuse the previous physical register allocated to the logical identifier for each shelf instruction's destination. We do not allocate new physical registers for shelf instructions, thus reducing the occupancy of the PRF.

Both shelf and IQ instructions translate their source register identifiers in the rename stage to pick up the physical register identifiers (PRI). They also pick up the existing destination register translation; the shelf simply uses it as a destination physical register, while the IQ will retire the identifier back onto the free list as it replaces the translation with a newly allocated physical register mapping. Figure 3.4 illustrates the life cycle of a PRI. A physical register is first allocated and written by an IQ instruction, and then overwritten by any number of shelf instructions until the

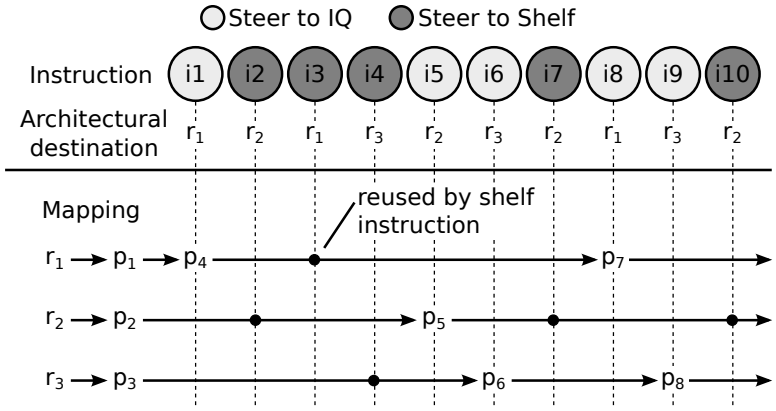



Figure 3.4: Life-cycle of register alias.

next IQ instruction renames the corresponding logical register and eventually retires it.

Instructions at the head of the shelf monitor a ready bitvector for their operand readiness (or may use pipeline interlocks like INO cores). The same method can be used to stall shelf instructions for WAW dependences. Nothing additional needs to be done for WAR dependences as the shelf issues in program order. Complications arise, however, when an IQ instruction waits on a true data dependence from an instruction on the shelf; the IQ cannot distinguish the potentially multiple writes to the same physical register by different shelf instructions, which all use the same PRI. In other words, there is an ambiguity in RAW dependences. Shelf instructions avoid this problem because they issue in program order. Once an instruction reaches the head of the shelf, only the last instruction to write a source operand may be outstanding, so there is no ambiguity. Dependent IQ instructions, on the other hand, join a dynamic instruction window, so they observe tag broadcast for multiple shelf writes to the same physical register. The rest of this section describes a mechanism to uniquely identify shelf writes to the same register for the IQ.

#### 3.2.3.1 Separation of Tag and Physical Register Index.

The problem at hand is that the PRI no longer uniquely identifies one instruction in the OOO window, as it does in a conventional PRF-based microarchitecture. Thus, a tag broadcast from one shelf instruction that writes a physical register might incorrectly wake up IQ instructions that depend on a different shelf instruction. To solve this problem while allowing shelf instructions to share a physical register, we must decouple the two traditional roles of the PRI as a destination register and as

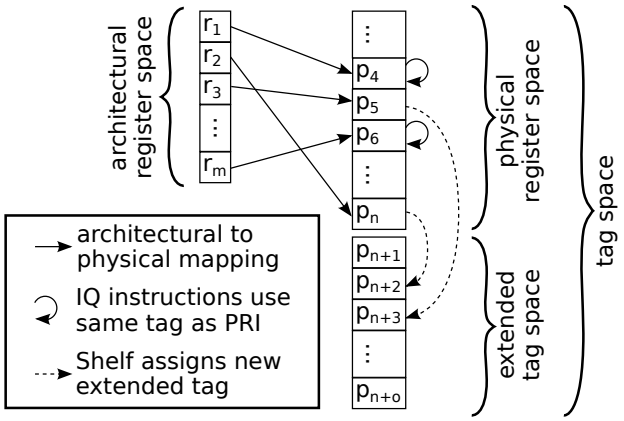



Figure 3.5: Extended tag space and mapping.

a unique identifier; each instruction acquires both a PRI and a unique tag from the rename stage. Thus an entry in the mapping table (MT) will now map an architectural register identifier to both a PRI and a tag.

Our implementation expands the tag space in a special way given the life-cycle of an architectural register. For IQ instructions, we retain the original tag space, where each tag corresponds to a particular physical register. When an IQ instruction allocates a new physical register, both its destination PRI and tag are set to that register's index. Shelf instructions allocate a new tag from an extended tag space without allocating a new register, and only change the mapping for the tag. We see that IQ instructions draw only from the original tag space, while shelf instructions draw only from the extended tag space. We manage these two portions of the tag space on separate free lists, one *physical* free list for the original tag space and one *extension* free list for the extension.

At rename, IQ instructions read the current mapping for their source operands, noting both the PRI and tag. The PRI is used to index into the PRF, and the tag is used to check readiness and for the wakeup operation. IQ instructions also pick up the current mapping for their destination registers, so as to retire the identifiers from the ROB to their respective free lists. The PRI is retired to the physical free list. If the current PRI and the tag differ, then the tag must be from the tag space extension and is retired to the extension free list. Finally, IQ instructions allocate a new PRI from the physical free list and set both tag and PRI mappings to it.

Shelf instructions similarly record all current mappings. At retire, they only return the tag to the extension free list if it differs from the PRI. Shelf instructions do not retire the PRI as the register remains in use and no new PRI is allocated. Only a tag is allocated from the extension free list, and used to broadcast to the IQ. We

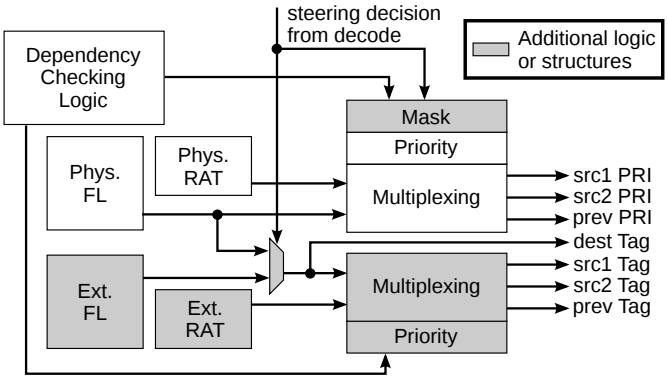



Figure 3.6: Extended rename stage.

illustrate these operations in more detail in Section 3.2.3.2 once we have described the steering mechanism.

#### 3.2.3.2 Rename Stage

Figure 3.6 depicts the extended rename stage. Steering is performed during decode, prior to rename, as steering decisions depend only on opcode and the architectural register names of operands and destinations. Depending on whether an instruction is steered to the shelf or to the IQ, its destination register and tag will be different. Tags from the extended tag space are offered by the extended free list (Ext. FL) and register alias table (Ext. RAT), while conventional PRI's are offered by their physical counterparts. The steering decision determines which structures are consulted to allocate a tag.

### 3.2.4 Memory Accesses and the LSQ

We first describe the ordering of shelf loads and stores under uniprocessor and relaxed/weak consistency models, which include the ARM v7 memory model used in our evaluation. Shelf loads and stores issue in program order, and thus follow all older loads and stores in the address calculation pipeline. As such, shelf loads and stores do not require their own load or store queue entries; instead, they record the tail pointers of both structures at dispatch to track their relative order.

Shelf loads associatively scan older IQ stores in the store queue, all of which have calculated their addresses and values, and younger IQ loads in the load queue, some of which may have been reordered and issued early to memory. (IQ loads perform the same operations as they execute). The shelf load receives the value from the youngest scanned instruction with a matching address. In particular, it must receive a value

from a *younger* matching load to avoid a memory ordering violation [47]. Loads with no matches issue to the cache hierarchy.

Shelf stores scan younger load instructions for matching addresses to perform store-to-load forwarding, or to squash IQ loads that have speculatively issued early. We use a "store sets" [12] memory dependence predictor to prevent frequent squashes. Shelf stores use their store set identifier to release dependent younger loads, just as IQ stores do. Finally, since uniprocessors and relaxed consistency models support coalescing store buffers and do not require ordering of stores to different addresses, shelf stores scan for the next older matching store and immediately coalesce into its store queue or store buffer entry. It is permissible to skip over older loads in this case because they will have already received a value from the coalescing buffer and taken their place in memory order (non-speculatively). Stores that find no match are released to the cache. We assume memory barriers synchronize the pipeline at the dispatch stage.

Stricter consistency models, like Total Store Order and Sequential Consistency, require in-window speculation [18] to enable high performance. Amongst other constraints, loads are speculative until all older loads to any address have at least completed (obtained a value from memory). As a consequence, all shelf instructions, including non-memory instructions, that follow a speculative load are speculative and may not writeback/retire until all preceding loads become non-speculative—an uncertain time interval (e.g., duration of a cache miss). Shelf stores additionally need to allocate store queue entries, as strong consistency models often do not permit coalescing in the store buffer. Evaluating the shelf under these models is beyond the scope of this paper. We suggest that steering mechanisms could steer those instructions to the shelf that are predicted to depend on long-latency misses, similarly to recent latency-tolerant designs [54, 21, 22].

## 3.3 Related Work

Hily and Seznec [23] show that the performance of an in-order core approaches that of an out-of-order core as the number of SMT threads increases, and argue that OOO cores are not cost-effective for SMT designs with many threads (four in their study). At the two extremes, OOO cores are suited to single-threaded workloads or those with few SMT threads, while workloads with a high number of threads favor in-order cores for efficiency. We reason that middle-range designs, which balance single-threaded performance and throughput, require a new underlying microarchitecture. We borrow

the name and concept for the shelf from the Metaflow architecture [49], which focused on the principle of shelving instructions to defer their execution, thereby enabling the out-of-order execution of other instructions. Ultimately, the Metaflow design was an OOO core centered on the DRIS structure, a combination of the ROB, IQ and renaming logic, used for all instructions. Our shelf physically separates in-sequence (deferred) instructions into a more efficient structure, while reordered instructions utilize the full capabilities of a modern OOO core.

Khubaib *et al.* rely on the same observations as Hily and Seznec to propose MorphCore [29], a design wherein the core can "morph" from an OOO with a low number of threads (two threads in their work) into an INO core with many (eight) threads. Whereas MorphCore offers a coarse-grain switching mechanism, our design enables the selection of OOO versus INO mechanisms on an instruction-by-instruction basis. MorphCore and our work target different objectives: MorphCore attempts to capture two workloads that do not often coincide, single-threaded and highly threaded, on one core; whereas, our design highlights an area where neither INO nor OOO cores are an efficient design point. Similar works provide a set of configurable cores by morphing, fusing or composing standalone cores [30, 25].

Viewed from another angle, our design attempts to approach the performance of a larger OOO instruction window through the use of in-order hardware. [52, 59] relieve the IQ by redirecting ready-before-dispatch instructions through energy-efficient functional units. Tseng and Patt [55] utilize compiler techniques to achieve a high performing schedule on in-order hardware, which approaches the single-threaded performance of OOO hardware. These designs, however, do not alleviate pressure on the ROB, LSQ and PRF. McFarlin, Tucker and Zilles [41] advocate similar designs by showing that OOO performance can be mostly achieved with static schedules, given the speculation support needed to permit those schedules. One such design is the in-order Continual Flow Pipeline (iCFP) [21], which targets long-latency operations like cache misses that block in-order cores. Miss-dependent instructions drain into a slice buffer, including any "side" inputs, to allow independent younger instructions to execute out-of-order. Drained instructions are re-executed from the slice buffer once the miss returns. In contrast, we steer instructions to OOO/INO up front (one-time execution). To enable correct out-of-order execution on iCFP, speculation is handled via checkpointing, which may be undesirable for SMT where the aggregate architectural state of all threads is much larger. Several other latency-tolerant designs [22, 54, 10] similarly rely on potentially expensive checkpoints; none of these designs leverage in-order hardware.

The shelf effectively reduces instruction occupancy in OOO structures. Several related works target similar goals without leveraging in-sequence instructions. Whereas there are many ways to reduce pressure on OOO structures, we note here those mechanisms most closely relate to our contributions. Elmoursy and Albonesi [14] reduce pressure on the IQ via predictive SMT fetch policies. Gonzalez *et al.* [19] reduce pressure on the PRF by decoupling tags (virtual registers) from PRIs (physical registers). Some works leverage checkpointing to release OOO resources early [40, 13]. Adaptive cores additionally provide the ability to disable unused structure entries [1, 48, 8].

Clustered microarchitectures divide the monolithic IQ structure across functional unit clusters to improve cycle time and scalability. Palacharla, Jouppi and Smith [45] advocate using FIFO queues in this manner to reduce complexity. Prior work focuses on steering instructions to clusters so as to minimize inter-cluster forwarding penalties and for load balancing [51, 3, 4]. Similar to our practical steering algorithm, these designs make use of dependence chain information for steering. While we do not cluster our execution units in this thesis, it is a possible dimension for the shelf and the IQ to belong to different clusters.

Several works examine heterogeneous cores [35] and datapaths [39]. These works fix a set of heterogeneous hardware resources, e.g., an OOO and an INO core, and attempt to schedule threads among them. Note that threads do not simultaneously use two heterogeneous components, but rather switch from one to the other. Numerous works propose scheduling schemes that target specific ILP/MLP regions [58], serializing bottlenecks in parallel code [27], and other indicators [34]. A number of these works advocate fine-grained switching at hundred- or thousand-instruction granularity [44, 39] but still fall short of interleaving in-sequence and reordered instructions in the same window. Our shelf and IQ datapaths can be seen as statically provisioned heterogeneous backends, however, a single-thread context is able to utilize both simultaneously, which is a central contribution of our microarchitecture.

# CHAPTER IV

# Steering

*With our hybrid microarchitecture in place, a steering mechanism is required to decide for each instruction whether it utilizes energy-intensive out-of-order hardware for increased performance, or energy-efficient in-order hardware. In this chapter we discuss ideal and practical steering mechanisms, and propose a predictive approach that can react to unexpected changes in the instruction schedule using simple hardware tables.*

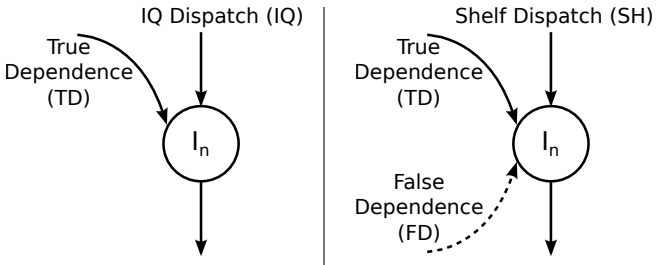
## 4.1 Overview

Instruction steering determines whether an instruction is dispatched to the IQ or the shelf, which directly affects the instruction schedule. Whereas the microarchitecture ensures correct execution under any steering policy, poor steering can result in poor performance. If we steer all instructions to the IQ, then the shelf provides no window size benefit. Conversely, if all instructions are steered to the shelf, the resulting performance will match that of an in-order core.

We base our discussion of steering mechanisms on a tractable, idealized steering algorithm. This allows us to define which decisions constitute a steering error by analyzing the instruction schedule after the fact. We then define a practical steering mechanism based on simple hardware components, which we show to achieve most of the benefit of the ideal algorithm, giving up only 1.1% of performance. Our evaluations examine steering errors and individual benchmark variations.

## 4.2 Ideal Steering

To guage the effectiveness of our practical instruction steering mechanism, we contrast it with the performance achieved by an ideal steering mechanism. Unfor-




(a) IQ instruction (b) Shelf instruction

Figure 4.1: Issue stage dependences depending on steering.

tunately, a perfect steering mechanism, which steers optimally for maximum performance, is a global optimization requiring complete knowledge of the whole-program critical path. Although mechanisms to predict instruction criticality have been proposed [17, 51], steering compounds the optimization problem: it adds/removes false dependence edges, which changes the very shape of the graph. Hence, even in the context of an offline oracle, perfect steering is intractable.

Instead, we contrast our practical steering against an ideal, local steering algorithm, which steers each instruction according to whether it would issue earlier from the IQ or the shelf (breaking ties in favor of the shelf). While this determination is made greedily without regard to future (younger) instructions, the greedy ideal steering algorithm requires precise knowledge of the future schedule. Such a mechanism cannot be implemented in practice, since these future arrival times are not always known at dispatch. However, in simulation, we can closely approximate this future schedule using complete knowledge of instruction latencies, dependences, and memory addresses. For memory operations, we functionally query the cache (atomically, instantly and not modifying state) to accurately predict memory latencies.

To formalize our ideal steering algorithm, we compare the dependence edges arriving at the issue nodes of IQ and shelf instructions in Figure 4.1. Both types of instructions have the same true dependences ($\overrightarrow{TD}$), but have different edges from the dispatch stage. Shelf instructions have shelf dispatch edges ($\overrightarrow{SH}$), while IQ instructions have IQ dispatch edges ($\overrightarrow{IQ}$). These edges carry dispatch stall cycles when their respective backend structures become full. Shelf instructions additionally have false dependences ($\overrightarrow{FD}$) arriving at the issue stage (see Section 2.3).

When an instruction is steered, for example, to the IQ, it does not stall on edges seen only by shelf instructions. However, given the state of the pipeline, we can measure whether these shelf-only edges *would have* arrived. At dispatch, an IQ instruction can query whether the shelf was full. Once its true dependences arrive, it

can query whether there are any remaining false dependences, which it need not stall for. Shelf-only instructions can similarly do this for IQ-only dependences.

$$\begin{aligned} & \overbrace{\max\left(T_{\overrightarrow{TD}}, T_{\overrightarrow{FD}}, T_{\overrightarrow{SH}}\right)}^{\text{Shelf issue cycle}} \leq \overbrace{\max\left(T_{\overrightarrow{TD}}, T_{\overrightarrow{IQ}}\right)}^{\text{IQ issue cycle}} \\ \leftrightarrow \quad & \max\left(\cancel{T_{\overrightarrow{TD}}}, T_{\overrightarrow{FD}}, T_{\overrightarrow{SH}}\right) \leq \max\left(T_{\overrightarrow{TD}}, T_{\overrightarrow{IQ}}\right) \\ \leftrightarrow \quad & \max\left(T_{\overrightarrow{FD}}, T_{\overrightarrow{SH}}\right) \leq \max\left(T_{\overrightarrow{TD}}, T_{\overrightarrow{IQ}}\right) \qquad \text{Ideal Greedy Condition} \quad (4.1) \end{aligned}$$

Since all relevant dependences arrive at the issue stage, we can easily compare whether an instruction would issue earlier from the shelf or the IQ, as in Equation 4.1. Notice that this determination is made with no regard to future instructions. In the case of a tie, it is always better to steer to the shelf, the only consideration made once an instruction has determined that it does not stall itself. Equation 4.1 frames these ideas in the Ideal Greedy Condition (IGC), where $T_{\overrightarrow{X}}$ is the arrival time of dependence edge $\overrightarrow{X}$. The left-hand side of the IGC corresponds to the expected issue cycle with the instruction steered to the Shelf, while the right-hand side corresponds to the issue cycle if the instruction is steered to the IQ. If the condition is true, our ideal steering algorithm steers to the shelf, otherwise it steers to the IQ.

Based on the IGC, we can now clearly define what steering decisions constitute an error. These are categorized in Figure 4.2. It is clear that we have an incorrect steering decision if it does not follow the IGC. In the case that the steering decision does match the IGC, we consider the following additional source of error in practical steering designs. A steering mechanism may prevent an instruction from dispatching for a number of cycles as it attempts to steer to a blocked side of the core. While this instruction stalls at dispatch, the steering mechanism may be able to obtain new information causing it to reverse its decision to a correct one. The blocked instruction proceeds to the correct structure so as to issue as early as possible (e.g. true dependences arrive later than the time spent blocking). The instruction appears to have been steered correctly, but in fact we consider this an error, which introduced unnecessary dispatch stalls for other instructions. Thus, a steering decision is correct if it matches the IGC condition, and allows the instruction to dispatch to the chosen side with no unnecessary delay.

Note that, because of the complexity of the gem5 simulation model, even with some oracle knowledge, we still steer an average of 4% of instructions incorrectly. Though it is highly detailed, our prediction of the future schedule does not account

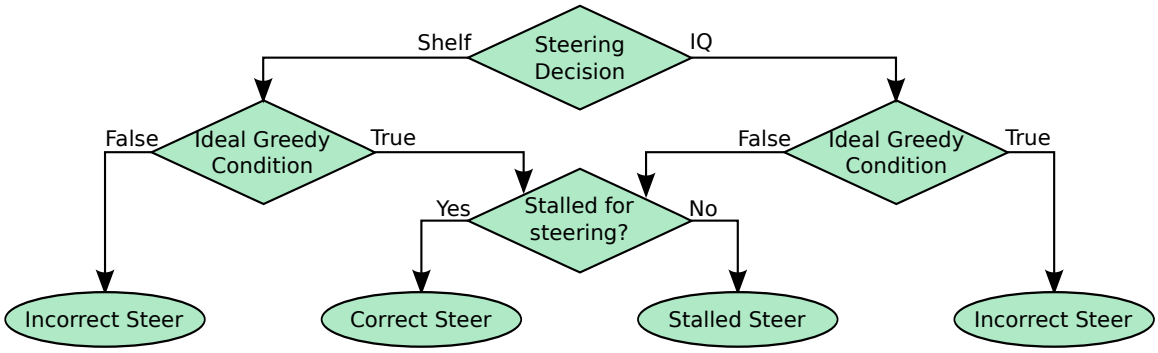



Figure 4.2: Errors under Ideal Greedy Steering.

for all corner cases that arise in the simulation. So, our algorithm additionally tracks the actual execution schedule as the simulation progresses to correct its representation of the schedule and recover from mispredictions.

## 4.3 Practical Steering

At the heart of steering lies (1) the ability to predict the future execution schedule, and (2) the ability to recover from schedule mispredictions. We describe a practical hardware solution to project instruction completion times and track dependence chains, and show how these mechanisms can be used for steering and misprediction recovery, with reference to Figure 4.3.

### 4.3.1 Schedule Prediction

For each architectural register, we maintain a prediction of its future writeback/ready cycle in a Ready Cycle Table (RCT). RCT entries are decremented each cycle to count down how many cycles are left until the register is predicted to be ready.

If we dispatch an instruction to the IQ, we can predict its issue cycle as the maximum ready cycle of its source operands, and its completion cycle as the issue cycle plus the instruction's predicted latency. Instruction latencies are usually available from decode. The prediction ignores structural hazards, such as issue width, and predicts all loads to be L1 hits; the resulting schedule errors are handled via the recovery mechanism.

If we dispatch an instruction to the shelf, it will issue after all previously dispatched instructions even if its operands are ready, since the shelf issues in program order. Hence, for the shelf, we maintain an earliest-allowable issue cycle, which is the maximum issue cycle of all previous instructions. Shelf instructions also must

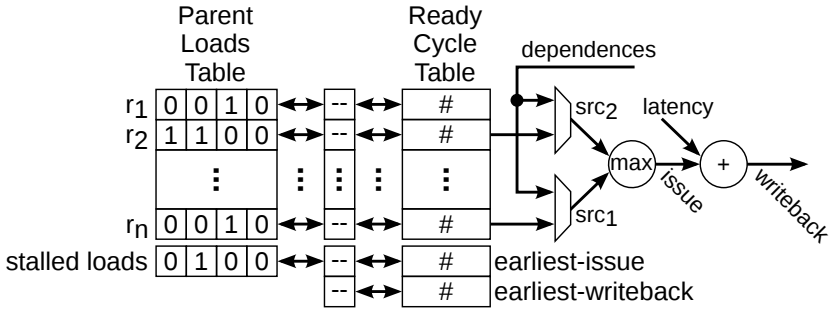



Figure 4.3: Practical steering.

stall at writeback while any preceding instruction is speculative. So, we also track an earliest-allowable writeback cycle, which is the maxmum speculation resolution cycle for any previous instruction. We can then predict that, if dispatched to the shelf, an instruction will issue at the maximum of its operands' RCT entries and the earliest-allowable issue cycle. Its completion cycle is predicted as the maximum of its predicted issue cycle plus the instruction latency and the earliest-allowable writeback cycle.

With these estimates, we can then steer an instruction by comparing its predicted completion cycle for the shelf and IQ, choosing the earlier of the two and breaking ties in favor of the shelf. Our design exploration shows that it is sufficient to track a range of 32 cycles using 5-bit counters per register.

### 4.3.2 Schedule Recovery

Our schedule prediction mechanism is approximate; most importantly, it assumes all loads are hits. As schedule errors accumulate, steering accuracy will worsen and performance will suffer. So, we correct schedule misprediction errors by observing the actual execution schedule and using the observed instruction completions to correct predictions for their dependent instructions.

Once a register's RCT counter decrements to zero, the register is predicted to be ready. However, if the instruction took longer than expected (e.g., an L1 miss), the register will not be marked ready in the issue dependency checking logic. In this circumstance, the predicted schedule for all the instruction's dependents is also incorrect. We correct these errors by freezing the decrement of the RCT entry for the destinations of all these dependents. We thereby push back the predicted completion time of the entire dependency tree by one cycle each cycle until the mispredicted instruction ultimately completes.

Maintaining RCT counters as we have just described requires tracking the dependency information among all instructions, which is expensive. Interestingly, we find that tracking the dependents of only a small sample of instructions is sufficient to correct the schedule; a schedule misprediction for an untracked instruction will rapidly be detected when one if its dependents is sampled. Since most schedule mispredictions are for loads that miss in L1, we track dependents for a sample of loads.

We use a simple bit matrix, the Parent Loads Table (PLT), to track the relationship between sampled loads and their dependents. As loads are steered, each is assigned a column of bits in the PLT, if one is available. Rows in the table correspond to architectural registers; a bit is set if the architectural register depends directly or indirectly on the load. When a load is steered, it sets the bit for its assigned column and destination register row. As further instructions are decoded, they set the row for their destination to the superset of their operands' parent loads (i.e., the bitwise OR of the operands' rows). When loads complete, they reset the bits in their assigned column, freeing the column for reuse. We find it is sufficient to track 4 loads per thread.

If any register's ready cycle reaches zero while its parent loads' bitvector is non-zero, we simply stall the decrement operation for all other registers which share those parents. The register's bitvector is loaded into a special row, the stalled loads bitvector, as shown in Figure 4.3, which is compared to all rows. Any row with a matching bit (i.e., it is directly or indirectly dependent on a stalled load) has its RCT counter stalled.

## 4.4 Alternative Steering Mechanisms

Our practical mechanism is both a predictive and reactive approach to instruction steering. We favored this approach due to the fine-grained interleaving of in-sequence and reordered instructions, as well as the highly non-deterministic schedules introduced by SMT. We discuss alternative approaches below but choose not to pursue them further as they provide a poor fit to our criteria.

- Making steering decisions statically without feedback at run-time. The steering mechanism can annotate instructions in the I-cache with some history information to guide the next decision. This static approach is geared towards machines where instructions are pre-scheduled by the compiler or the front end, as opposed to the dynamically changing issue schedules introduced by SMT.

- Targeting important scheduling events, like long-latency memory misses, similarly to latency-tolerant designs [21, 22, 54, 10]. While this approach benefits from leveraging existing techniques which have been demonstrated in the literature, events such as long-latency misses are relatively coarse-grained and do not lend themselves to the fine-grained, instruction-by-instruction capability of our microarchitecture.

- A history-based approach, based on the notion that OOO schedules tend to repeat [41]. This method could use design principles from branch predictors to learn and store information from past instruction schedules. As with the static approach, this method is complicated by the dynamic SMT issue schedules, requiring that the history information account for potential deviations from the previous schedule.

Steering can additionally interact with SMT fetch and issue policies. Just as SMT can make decisions on the thread-level, steering can prioritize one thread over another at different intervals. High-priority threads can be selected for a number of reasons, including fairness, performance or criticality. In our evalutaion, steering as well as SMT fetch and issue treat all threads equally. It is beyond the scope of this thesis to explore the interaction of steering with asymmetric SMT policies.

# CHAPTER V

# Evaluation of Hybrid Core

*We evaluate the shelf under both practical and idealized mechanisms and demonstrate that it can nearly double the instruction window while respectively offering 8.3% and 12.5% improvement in energy-delay product over the best-sized out-of-order baseline.*

## 5.1 Methodology

We model our design in gem5 [7] and run the SPEC CPU2006 benchmark suite with the ARM v7 ISA using system call emulation. We have excluded only *dealII* of the 29 SPEC benchmarks as it is not functional in our simulation infrastructure. For N-thread SMT workloads, we generate 28 benchmark mixes consisting of N different SPEC benchmarks, such that each benchmark appears in N different workloads. Using the reference input set, we fast forward all threads for 4 billion instructions, before warming up microarchitectural structures for 200 million instructions, and then report on the next run of 200 million instructions (per thread) in detailed simulation.

Table 5.1 details our configuration. We assume a 2 GHz clock for all configurations to focus on the microarchitectural effects of our technique. Unless otherwise stated, our evaluation focuses on a 4-thread SMT configuration using the ICOUNT fetch policy [56]. The ROB, load queue (LQ) and store queue (SQ) structures are partitioned across threads, based on [33], as are the front-end pipeline buffers and the shelf to prevent stalled threads from blocking others. We represent a core size with the number of ROB entries, swept in powers of 2 from 32 to 256 entries. The other OOO structures, including the IQ, LQ and SQ are proportionally scaled at half the ROB size (16 to 128). The shelf, when present, has the same number of entries as the ROB. We primarily contrast a system with an N-entry ROB and shelf against a conventional OOO core with 2N ROB entries.

| Component | Configuration |
|---|---|
| Core | 4-thread SMT OOO @ 2.0 GHz<br>4-wide OOO with 8-wide fetch<br>6 cycles fetch-to-dispatch |
| ROB | Variable (32, 64, 128, 256) |
| IQ, LQ, SQ | Half of ROB |
| Shelf | Same as ROB |
| L1I | 32KB, 2-way, 1-cycle |
| L1D | 32KB, 2-way, 2-cycle |
| L2 | 2MB, 8-way, 32-cycle |
| Memory | 100ns latency |

Table 5.1: System Configuration

For power and energy analysis, we use the McPAT framework [36] to model the power breakdown of a physical register-based OOO design, incorporating changes from [37]. We modify McPAT to incorporate the shelf, by modeling this new structure with the same components as the ROB. Steering hardware is assumed to consist of a RAT-sized structure in the front end. We report on the power consumption of the core including L1 caches.

## 5.2 Impact of the Shelf

The shelf extends the instruction window for in-sequence instructions, improving performance. At some point, however, additional shelf entries no longer provide the benefits of a larger instruction window, as additional instructions steered to the shelf begin to stall on false dependences. For additional benefit, an increase in baseline OOO hardware resources is required to expose additional ILP. Thus, the shelf provides some size increase or *size-up* for every baseline OOO to which it is added. We expect the effectiveness of the shelf to vary with the fraction of in-sequence instructions seen at a particular target window size and number of SMT threads.

We sample this cross-product of design points in Figure 5.1. Single-threaded and 4-thread core configurations are shown in Figures 5.1a and 5.1b respectively. We report performance (left), energy-efficiency (middle), and energy-delay product (right) for each configuration. The shelf is evaluated here with idealized steering (ideal) for both the conservative issue stage mechanisms (cons.) as well as aggressive ones (aggr.) with no cycle penalty. The conservative mechanisms are discussed in

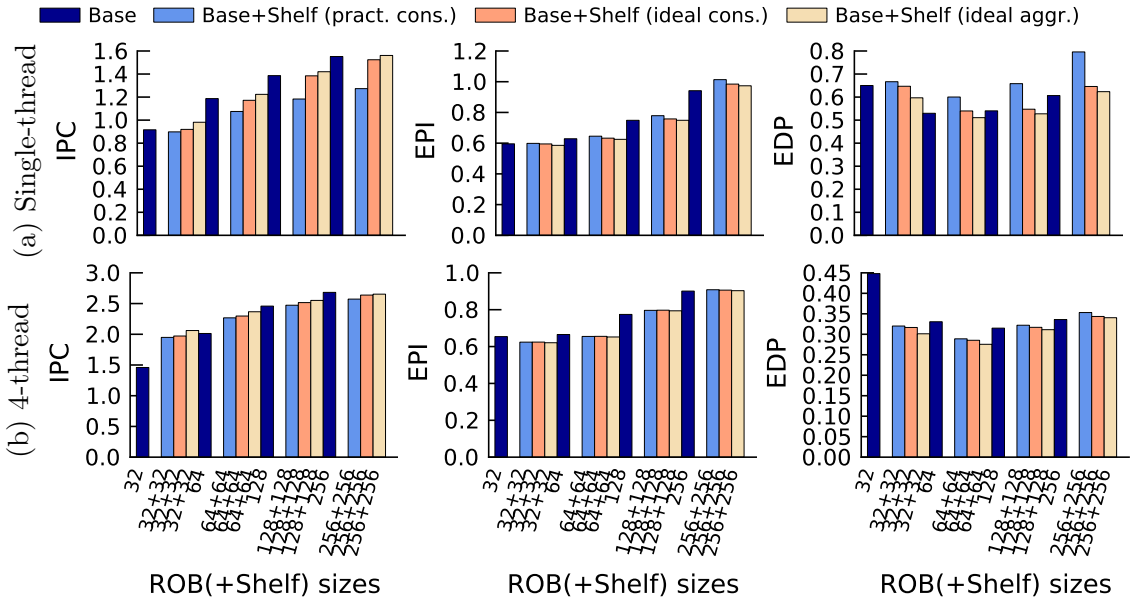



Figure 5.1: Comparison of doubling OOO structures against adding a shelf.

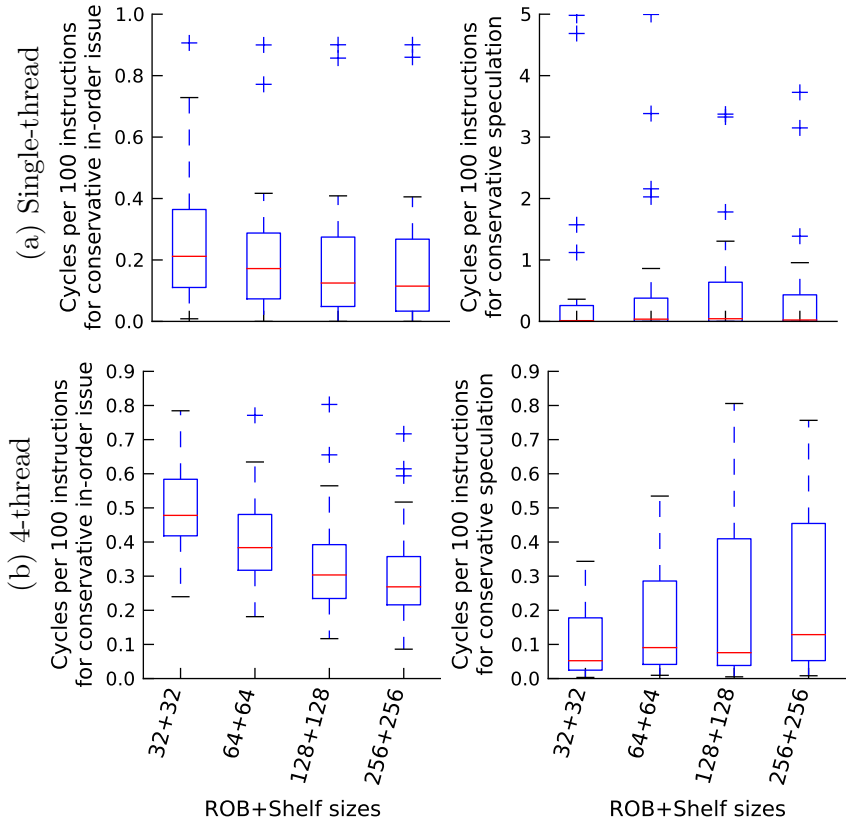



Figure 5.2: Potential stall cycles due to conservative mechanisms.

Section 3.2.1.1 and Section 3.2.2.1. We also evaluate our practical steering mechanism with conservative hardware (pract. cons.).

**Size-up.** We find the shelf is more effective for 4-thread cores than single-threaded cores, as predicted by the trend in Figure 2.5. For 4-thread cores, the shelf approximates the performance of doubling the OOO instruction window, but begins to fall short as the baseline size increases. This falloff happens for two reasons: (1) the fraction of in-sequence instructions falls with increases in OOO resources, and (2) extending the OOO window itself yields diminishing returns, given the same functional unit pool.

**Energy-Efficiency.** The shelf roughly maintains the energy-efficiency of the baseline OOO it augments—the power overhead of the shelf is roughly offset by its performance improvement.

**Energy-Delay Product.** We quantify the distance between designs in terms of energy-delay product, and focus the rest of our evaluation on the best two points: the baseline OOO with 128 ROB entries, and our design with 64-entry ROB and shelf. Overall, the conservative shelf design reduces the energy-delay product by 9.4% (lower is better), averaged across all workloads (geometric mean), over the best baseline OOO size. The aggressive design reduces the energy-delay product by a *further* 3.1%.

## 5.3 Impact of Conservative Mechanisms.

We quantify the effect of each conservative hardware configuration. Figure 5.2 shows the issue slots wasted by conservative mechanisms for single-threaded and 4-thread cores. For every core size, a box plot depicts the distribution across workloads. On the left, we show the slots wasted due to shelf instructions not being allowed to issue in the same cycle as an older IQ instruction (issue-tracking bitvector update is on the critical path). For both single-threaded and 4-thread cores, this never exceeds one slot per 100 instructions. On the right, we show the slots wasted due to shelf instructions stalling for the speculation of younger IQ instructions (only two SSRs as opposed to per-run or infinite SSRs). While 4-thread cores still never exceed one slot per 100 instructions, there are a handful of single-threaded outlier benchmarks which go as high as 5 slots per 100 instructions. This suggests the need for more SSRs for single-threaded benchmarks; nevertheless, the potential for speedup is bounded by the aggressive design.

## 5.4 Impact of Practical Steering

We find that our algorithm provides adequate steering with 5-bit RCT entries and a 4-load PLT. We have modeled these structures in McPAT with a RAT-sized table.

In the 4-thread cores, our practical steering algorithm comes close to achieving the performance of ideal steering. It only loses 1.1% in energy-delay product, which leaves an 8.3% advantage relative to the best baseline core. Recall that ideal steering with conservative hardware achieved a 9.4% energy-delay improvement. Thus, most of the opportunity to improve the shelf design comes from mitigating the conservative mechanisms of the backend (an additional 3.1%, see Section 5.2), and not from steering.

## 5.5 Individual Benchmarks

On the other hand, single-threaded benchmarks suffer under practical steering. We examine individual benchmarks in Figures 5.3 and 5.4 to uncover variability in the shelf's behavior. We find that a handful of single-threaded benchmarks are particularly detrimental to our practical shelf design when comparing with the idealistic shelf. These include *bwaves*, *GemsFDTD*, *lbm*, *leslie3d* and *sphinx3*. These benchmarks suffer from a relatively large fraction of instructions steered to the shelf erroneously. On the other hand, 4-thread benchmarks give rise to relatively stable shelf performance characteristics, as errors in one thread are compensated by parallelism from the others.

## 5.6 Steering Errors

We finally focus on the steering breakdown of practical steering mechanism relative to that of the ideal greedy steering algorithm. We run the ideal algorithm alongside our practical mechanism (which drives the schedule) in simulation, and record both steering decisions. For every shelf or IQ steer of the ideal algorithm, we characterize our practical mechanism's decisions as hits or misses in Figure 5.5. Although 16% of instructions are steered "incorrectly", we find that the performance impact is relatively small.

## 5.7 Summary

Whereas OOO execution can improve performance for moderately threaded SMT designs, the resulting hardware utilization is inefficient, as many instructions are scheduled in-sequence. We have described a new microarchitecture that augments an OOO core with an energy-efficient in-order scheduling mechanism, the shelf, which allows in-sequence instructions to interleave correctly at fine granularity with re-ordered instructions. This microarchitecture achieves 8.3% improvement in energy-delay product, extending the performance-energy pareto frontier beyond baseline OOO core designs.

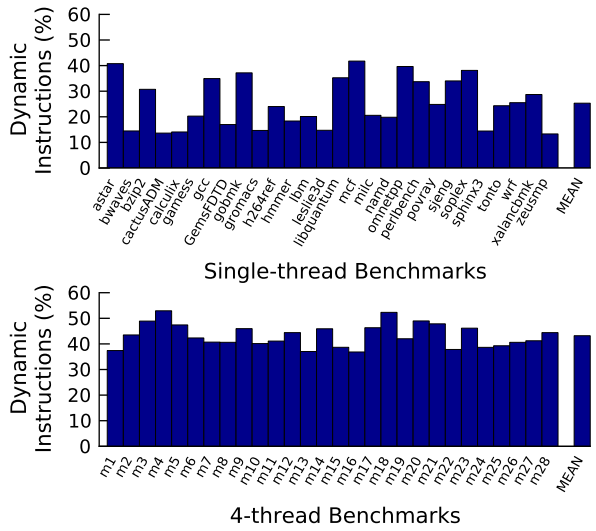



Figure 5.3: In-sequence instructions in Base(128).

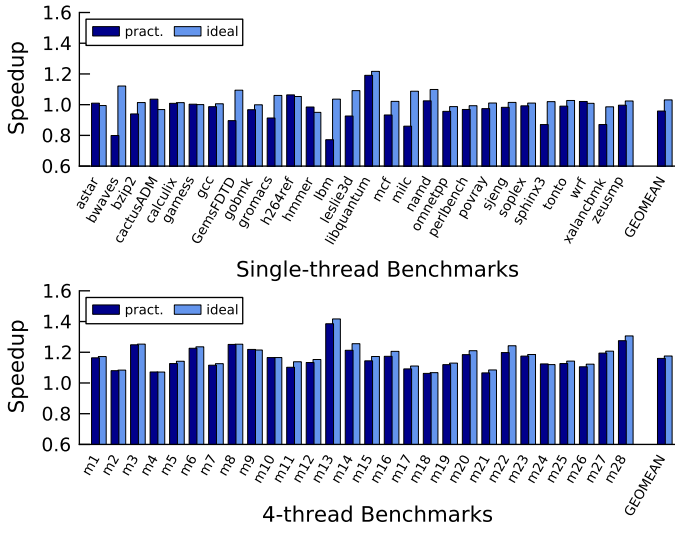



Figure 5.4: Speedup of Base+Shelf(64+64) relative to Base(64).

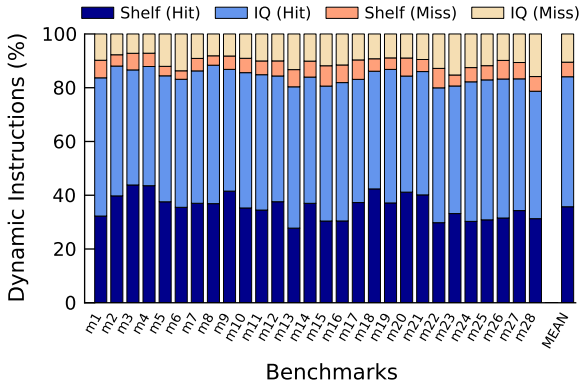



Figure 5.5: Steering hits/misses compared with ideal greedy algorithm.

# CHAPTER VI

# Embedded Way Prediction for Last-Level Caches

*This chapter investigates* Embedded Way Prediction *for large last-level caches (LLCs): an architecture and circuit design to provide the latency of parallel tag-data access at substantial energy savings. Existing way prediction approaches for L1 caches are compromised by the high associativity and filtered temporal locality of LLCs. We demonstrate: (1) the need for wide partial tag comparison, which we implement with a dynamic CAM alongside the data sub-array wordline decode, and (2) the inhibit bit, an architectural innovation to provide accurate predictions when the partial tag comparison is inconclusive. We present circuit critical-path and architectural power/performance studies demonstrating speedups of up to 15.4% (6.6% average) for scientific and server applications, matching the performance of parallel tag-data access while reducing energy overhead by 40%.*

## 6.1 Background

**Types of cache access.** Latency-sensitive L1 caches typically adopt a *parallel* access scheme, shown in Figure 6.1 (left), wherein all ways of both the tag and data array are read concurrently, thus minimizing latency at the expense of energy efficiency. Conversely, L2 or LLC designs typically perform tag accesses first, followed by an access to the correct data way, as depicted in the *sequential* access scheme in Figure 6.1 (center). Sequential access saves the energy of accessing irrelevant data ways in larger, higher associativity caches, at the cost of latency. *Way prediction*, illustrated in Figure 1 (right), attempts to offer the best of both, by only accessing a subset of data ways in parallel with the tags.

We highlight two special cases of way prediction. The first is *way filtering*, where the cache access is nominally parallel, but data ways that are known not to contain the requested block are filtered out. Way filtering never incurs a performance

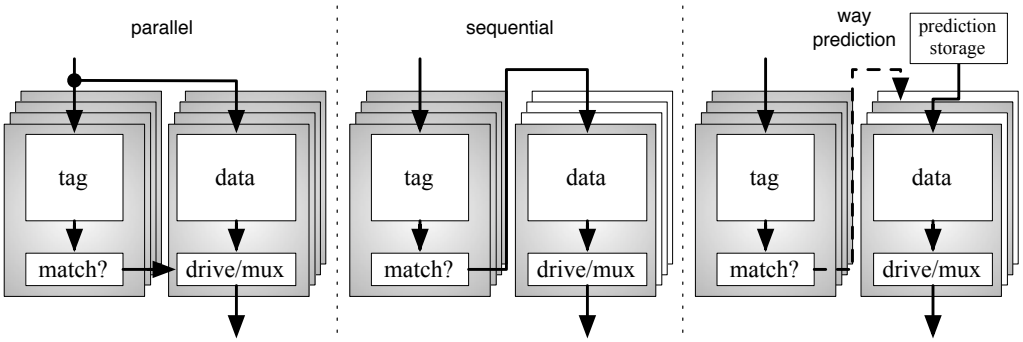



Figure 6.1: Cache access schemes.

overhead relative to parallel access, but may consume as much energy if filtering is not successful. The second is *single-way prediction*, where at most one data way is activated in parallel with the tags. We focus on this flavor of way prediction, as it assures that at most two data ways will be activated per cache access, bounding LLC access energy. In the common case that the prediction is correct, single-way prediction offers the low latency of parallel access at the low energy of sequential access. On a misprediction, the tag comparison triggers a second, sequential data access.

**Partial tag matching.** Although predicting the most-recently-used (MRU) way at the L1 is known to be 85-95% accurate [9], we find it is typically only 30-60% accurate at the LLC and sometimes little better than a random guess for multithreaded scientific and server workloads (see Section 6.5.3). Instead, to quickly and efficiently rule out ways that cannot contain the requested cache block, we advocate comparing the low-order bits of each stored tag to the incoming address, an idea known as *partial tag matching*. In the context of parallel caches, partial tag matching can implement way filtering by inhibiting access to ways that mismatch. Prior designs using partial tag matching in this manner have targeted L1 caches, as parallel access is not typically used in LLCs. The key challenge in such designs is engineering the partial tag match so that it has minimal impact on the data array critical path while saving as much energy as possible.

## 6.2 Related work

Our work builds on a long history of literature on way prediction, way filtering, and partial tag comparison; the earliest work in this area dates back over 20 years [28]. Broadly, our objective is to revisit these concepts in the context of modern servers

because of their growing sensitivity to LLC access times. Our design accelerates LLC tile accesses in the common case of an accurate way prediction.

Way prediction was initially proposed as a performance enhancement to prearrange multiplexor paths at the output of cache data arrays to select the MRU way before the tag comparison is complete [38]. Subsequent work suggested using sources besides the replacement order, such as register or instruction addresses, to predict which way to access first in sequential associative caches (which access cache ways in consecutive cycles) [9]. Later work focused instead on saving energy by accessing a single predicted way in parallel caches [24, 50], predicting wake-up for Drowsy cache cells [31], or selective sub-array precharging [62]. In all of these designs, a key constraint is that the prediction must be made before the cache address is available, a constraint relaxed in LLCs.

Partial tag matching was first suggested by Kessler and co-authors to reduce the number of tags scanned sequentially in early set-associative caches, where tag comparators were expensive [28]. Over the past two decades, partial tag matching has been suggested in various forms as a means to reduce tag comparison energy in sequential caches, most recently using a partial tag bloom filter [46]. Min and co-authors use a partial tag match to gate sense amplification and bit line muxing [42]. Zhang and co-authors conserve nearly all of the data array access energy by performing the partial tag match in parallel with wordline decode, then gating wordline activation [63]. We pursue the same approach. However, whereas their study targets a small (8KB), low-associativity (4-way) L1, we target a comparatively massive (2MB) highly-associative (16-way) LLC tile in a multicore server, leading to a markedly different solution.

## 6.3 Architectural Design

We propose *Embedded Way Prediction* in the context of a server-class chip-multiprocessor with a highly-associative large tiled last-level cache similar to designs from Tilera [5]. We consider both private and shared cache organizations. Similar to [63], we perform partial tag matching by embedding a CAM alongside the wordline decoders of the data SRAM sub-arrays. However, we find server workloads require a far wider partial tag comparison of 6-8 bits (see Section 6.5.3), which necessitates a dynamic CAM circuit to avoid timing impact. Furthermore, we target single-way prediction as opposed to way-filtering to limit energy per access.

### 6.3.1 Addressing Partial Tag Collisions

Because the partial tags are narrower than full tags, it is possible for the partial tags to match in several ways. These *partial tag collisions* lead to ambiguity as to which one among the matching ways should be predicted. To avoid the energy overhead of multiple way accesses (only one of which can be correct), we instead include an *inhibit bit* in each CAM entry that, when set, prevents that entry from reporting a match. We orchestrate the inhibit bits such that they are set for all but one colliding partial tag, and also use them to disable matches for invalidated lines.

A variety of policies might be used to choose which among a set of colliding tags should remain enabled. In our design, we use our LRU replacement policy as our guide, and clear the inhibit bit for the MRU tag within each collision set. We explore the impact of this scheme on prediction accuracy in Section 6.5.3. Inhibit bits could also be used for more complex schemes (e.g., using information from more sophisticated replacement policies [26] or confidence counters) or to provide software control over the use of embedded way prediction (e.g., to activate it only for blocks allocated by a particular core/thread). We note that embedded way prediction has no effect on the cache replacement algorithm or coherence protocol.

### 6.3.2 Maintaining Inhibit Bits

To ensure that only the MRU block within each collision set can trigger a parallel lookup, we maintain the following invariant: each time a cache block within a set is accessed or newly allocated (when a miss is filled), its inhibit bit is cleared, while the inhibit bits for any other block matching the same partial tag are set. Given this, at most two inhibit bits can change per cache access.

To avoid the need for a read port on the CAMs, the tag array stores a copy of the inhibit bit for each way (1 bit of overhead for every 32-bit tag). Rather than calculate and update inhibit bit state within the embedded way predictors at the data array, we instead rely on the tag array to maintain their state, sending updates to the CAMs when needed. We reuse the low order bits of the tag array's tag comparator to identify matching partial tags. With the information stored in the inhibit copies, we can identify which way was predicted within the data arrays. We can also identify if the prediction was correct, by checking it against the full tag comparison. At the tag array we then set the inhibit bit for all matching partial tags, except that we clear the inhibit bit for the hit way (if the access was a hit). The new inhibit bit state is

driven to the data arrays along with the way select signal, and modified inhibit bits are written into the appropriate CAM entry.

On a misprediction, the (sequential) access to the correct way within the data arrays must override the partial tag comparison to ensure the word line is activated. Rather than add an override input to the CAM/decoder circuits, which would impact the critical path of one (or both), we solve this problem architecturally, by driving the partial tag comparison and inhibit match lines with the (known) content of the CAM. We make use of the fact that the inhibit bit already takes part in the CAM comparison: during a way prediction we clear the comparison lines to inhibit bits so that cleared inhibit bits can match. Now, during the sequential access, the comparison line is set for the hit way to force a match, while the rest of the ways are disabled by setting their inhibit bits to the opposite of their known values.

Finally we address cache replacements and invalidations that target the MRU matching partial tag. Depending on the LRU implementation, these events make identifying the next-most-recently-used matching partial tag ambiguous (for example, an approximate LRU implementation). In these scenarios, we clear the inhibit bit of an arbitrary other matching partial tag. We see in Section 6.5.3 that the potential impact of a wrong choice in this situation is low.

### 6.3.3 LLC Tile Organization

Each LLC tile in our design is 2MB, 16-way set associative, and divided into 4 independently operating banks. Within a bank, tag and data pipelines are separately scheduled, to facilitate coherence traffic that often requires only tags. Each 512KB bank contains 512 sets of 16 ways each with 64B blocks. Within a bank, the tag and data arrays are further sub-divided into sub-arrays to optimize the latency-area-power trade-off. We assume a physical layout like that modeled by CACTI [43], where tag and data sub-arrays are grouped into *mats*: two-by-two squares of sub-arrays that share a common predecoder. The mats are interconnected via intra-bank H-trees, which in turn connect to the bank-level routing. As these interconnect busses are long and wide, they account for the majority of cache dynamic power (and, to a lesser degree, latency) for sequential accesses.

We organize ways across mats such that each 32KB way occupies two 256x523 bit sub-arrays (512 data + 11 ECC) from the same mat. Our data arrays rely on ECC rather than bit interleaving to provide tolerance against soft error. Otherwise, bit interleaving would complicate partial tag matching because several data words (with different tags) share a single wordline. Hence, the way predictor would have

to activate the wordline if any of the corresponding tags matched, requiring an OR function in addition to several CAM comparisons (one per interleaved word).

An arriving read that finds the data pipeline unscheduled forwards its partial tag and set index to all data mats/ways to initiate a way prediction. This in turn activates the CAMs and decoders for all data ways, which proceed to read out at most one uninhibited block with a matching partial tag. In the meantime, the result from the full tag comparison is used to confirm the prediction. On a misprediction, the correct data way is then activated, thus incurring an extra data way access relative to a sequential cache.

## 6.4 Circuit design

We study the effect of partial tag width on LLC way prediction accuracy in Section 6.5.3, and determine the need for 6-8 bits to achieve an accuracy over 90%. The crucial question then is to determine whether such a wide partial tag comparison can be hidden within the wordline decoder delay and to determine the energy requirements of the CAMs themeselves. In this section, we perform a critical path analysis of the partial tag comparison and wordline decoder circuits. We first describe the physical layout of our LLC bank, and finally provide energy estimates for parallel, sequential, and way-predicted cache accesses.

### 6.4.1 Data Mat Organization

Figure 6.2 shows the internal layout of a single data mat, including the additions required to support embedded way prediction. In a conventional sequential access, the tag arrays indicate which data way to access, and one of the sub-arrays within a single mat returns the data. The 9 index bits, along with an enable signal, arrive via the intra-bank H-tree at the predecoder, which comprises four 2-4 decoders, one of which is gated by the enable, a NAND combining stage which creates a one-hot encoding of the combined outputs of a 2-4 decoder pair, and drivers to transmit the predecoded index to the sub-arrays. A final NAND stage combines the predecode signals into wordlines, which are driven across the sub-arrays.

To support embedded way prediction, each data sub-array within the mat is augmented with a 256-entry CAM. Because a CAM cell requires two horizontal routing tracks, while modern SRAM cells have only one, we place two separately-driven 128-entry CAM arrays (top CAM and bottom CAM) side-by-side, with each CAM row spanning two SRAM rows. Note that the match line of one CAM must route

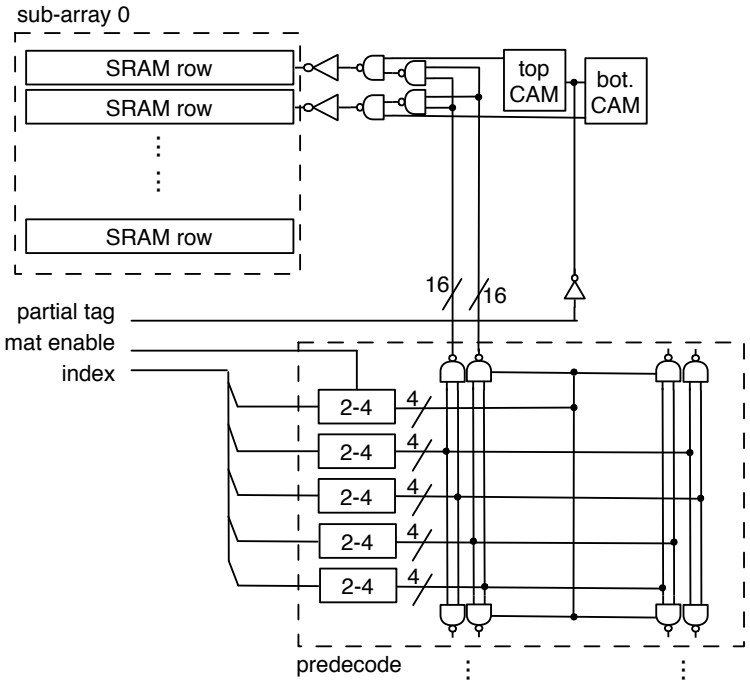


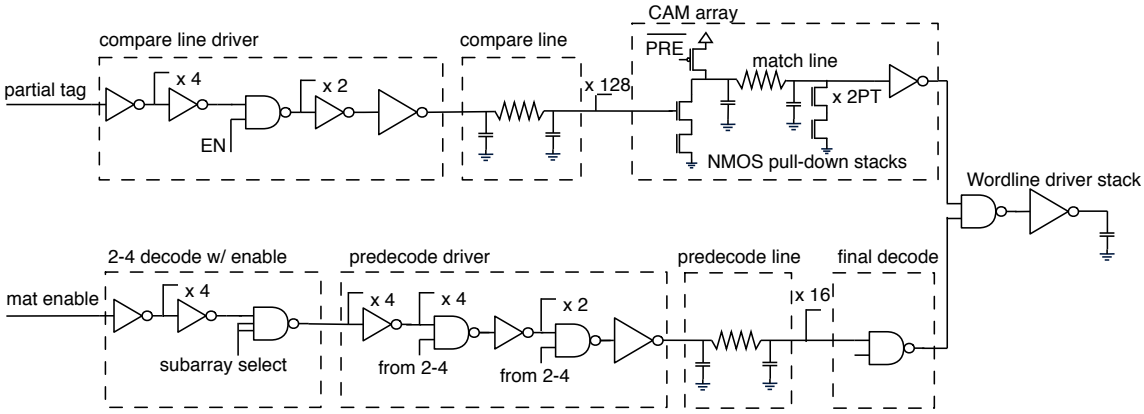

Figure 6.2: Organization of a single data "mat".

over the other, requiring resources on an additional metal layer. When predicting, all 4 CAMs (in each of the 16 ways) operate in parallel with the mat predecoder and sub-array wordline decoders.

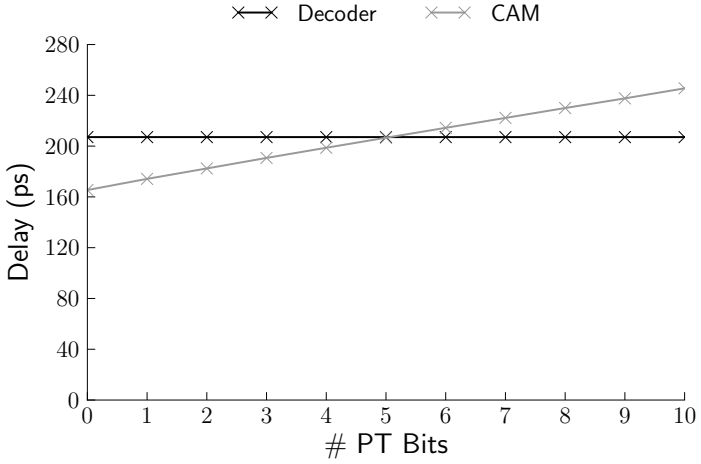
The need for wider comparison forces us to use a dynamic CAM circuit [32], rather than using the transmission-gate XOR-NOR comparator proposed in [63]. The 10T CAM cell performs the (mis)match operation by pulling down a precharged matchline through an NMOS stack. Indeed, Zhang et al. considered a dynamic CAM and rejected it because their static logic design is more energy efficient for small sub-arrays; we reach the opposite conclusion for LLCs.

### 6.4.2 Critical path analysis

Our objective is to establish the limit on the number of partial tag bits that can be compared within the timing constraint of the wordline decoder circuit for our chosen configuration. We optimize the critical paths of the decoder and CAM circuits using the method of logical effort, and investigate their timing using Cadence Virtuoso Spectre targeting an industrial 65nm process. For our target technology, the SRAM bit cell is 1.05$\mu$m by 0.46$\mu$m, and the CAM cell is 1.05$\mu$m by 0.92$\mu$m. We stop short of analyzing a complete mat layout, as this would need to consider additional factors




(a) CAM (top) vs. Decoder (bottom) paths.




(b) Delay comparison.

Figure 6.3: Critical path analysis.

beyond the scope of this study, including component-level energy trade-offs, reliability under process variations, and peripheral logic unrelated to embedded way prediction.

We begin our analysis with the assumption that the mat input signals arrive simultaneously. From there, the decoder and CAM circuits follow different critical paths before converging at the wordline driver stack, as depicted in Figure 6.3a. Optimizing decoder critical paths is discussed extensively in [2]. We only note here that the critical path of a decoder is complicated by the intermediate wire load of the predecode lines. We simplify the optimization process by following a heuristic adhered to in CACTI and suggested in [2], namely, to set the NANDs after the predecode lines to minimum size, thereby improving the energy-delay characteristic of the decoder. The resulting critical path of our decoder, illustrated in Figure 6.3a (bottom) yields a decode time of 207ps for our 65nm process.

On the other hand, our CAM operates by broadcasting each bit to be compared

| | Sequential | Way-Predict | | Parallel |
|---|---|---|---|---|
| | | Corr. | Mispr. | |
| **Routing** | 0.8594 | 0.8594 | 0.8594 | 0.8594 |
| **H-Tree** | 0.5470 | 0.5470 | 0.5470 | 0.5470 |
| **Decoder** | 0.0003 | 0.0045 | 0.0047 | 0.0045 |
| **Subarrays** | 0.1016 | 0.1016 | 0.2033 | 0.8130 |
| **Tags** | 0.0119 | 0.0119 | 0.0119 | 0.0119 |
| **CAM** | - | 0.2051 | 0.2051 | - |
| **Total** | 1.5203 | 1.7295 | 1.8315 | 2.2359 |

Table 6.1: Per-Access Dynamic Energy in nJ.

(along with its inverse) to all rows. Since the highly regular CAM cells are constrained by stringent sizing and layout considerations, rearranging logic within the CAM cell is not possible. At best, the comparison line driver can be optimized to reduce the delay on the comparison line. Thus, as we increase the number of partial tag bits in the CAM, the load on the NMOS stack (which cannot be sized to compensate) also increases, while each comparison line driver continues to drive the same load.

Therefore, we see the delay of the CAM circuit degrade in Figure 6.3b with increasing number of partial tag bits (CAMs also include one extra inhibit bit cell). Our results show that a CAM of width $5 + 1$ can be designed within the timing constraint of the decoder. However, even for slightly wider CAM widths, which are desirable for the embedded way prediction architecture, the CAM delay does not exceed that of the decoder by more than 20ps. For a CAM including 7 partial tag bits, needed for accurate way prediction at the LLC (Section 6.5.3), such an overhead is negligible compared to the overall cache access time (several nanoseconds).

### 6.4.3 Energy per Cache Access

Finally, we estimate the energy per access of sequential, parallel, and way-predicted (correct and mispredict) accesses, using estimates obtained from CACTI 6.5 and our characterization of the embedded CAM circuit using Spectre. We resort to CACTI, rather than using the estimates available from an SRAM compiler, because CACTI provides a detailed energy breakdown across components of the cache bank, while the available SRAM compiler provides only a black-box total. We require the breakdown to accurately assess the overheads of embedded way prediction.

We configure CACTI 6.5 to target the 65nm ITRS-LOP process and constrain CACTI's search to generate a 2MB 4-bank cache with a physical layout conforming

| Component | Configuration |
|---|---|
| Cores | 16 OoO Cores @ 4.0 GHz<br>8-stage pipeline; 4-wide OoO<br>96-entry ROB, LSQ |
| Architecture | Ultra Sparc III ISA |
| L1I Caches | 64KB, 2-way, 64B blocks |
| L1D Caches | 64KB, 4-way, 64B blocks, 32 MSHRs |
| LLC Cache | Tiled, 2MB per-core private |
| LLC Tiles | 16-way, 64B blocks<br>15-cycle parallel, 21-cycle sequential |
| Interconnect | 2-D folded torus, 2-cycle router<br>1-cycle link latency |
| Directory | MOESI coherence<br>8K entries per tile (128K total)<br>16-way, 4-cycle latency |
| System Memory | 3GB, 4KB pages, 150 cycle latency |

Table 6.2: System Configuration & Workloads.

to the organization described in Section 6.3.3. We report the resulting energy breakdowns in Table 6.1. We categorize the various components into inter-bank Routing, intra-bank H-Tree, data row Decoders, data Subarrays (precharge, bitlines, sense amps, and array-internal muxing and output drivers), Tags (all sub-components), and CAM (all CAMs within a bank). The sequential access energy breakdown is taken directly from CACTI; the other estimates are formed by multiplying per-element energy by the number of elements activated during the given type of access. Based on CACTI's access time estimates, we determine a parallel access latency of 15 cycles and a sequential access latency of 21 cycles for a 4GHz clock assumption.

Like many large caches, the energy consumption of our design is wiring-dominated, except in the case of parallel access, where the concurrent accesses to all 16 data ways dwarf all other components. Overall, the per-access overhead of a successful way prediction is only 13.8%, and a misprediction is 20.5%. In contrast, a parallel access incurs 47.1% more total energy. We use these estimates to construct the full LLC power and energy estimates in Section 6.5.

## 6.5 Evaluation

We first establish the effectiveness of our final, tuned embedded way prediction design with 7 partial tag bits (denoted 7) relative to sequential (S), parallel (P), and per-set MRU-way-prediction (M) baselines. We then study the sensitivity to partial tag width, the impact of partial tag collisions, and the importance of including inhibit bits in the design. Our designs are denoted by the number of partial tag bits, and an additional inhibit bit is present in all cases. Finally, we examine energy and power implications.

### 6.5.1 Methodology

We evaluate the architectural impact of embedded way prediction on a suite of scientific and server applications using the Flexus full-system simulator [60]. We configure our processor model to approximate the hardware resources of recent Intel Xeon microarchitectures; details appear in Table 6.2. We simulate a 16-core tiled chip multiprocessor with 32MB aggregate on-chip LLC capacity, composed of 16 per-core 2MB tiles with MOESI coherence. To study the generality of embedded way prediction, we examine two baseline organizations: 1) a *private* organization where each core queries and allocates blocks in its local tile, and 2) a unified *shared* organization where the address space is interleaved across tiles and a block resides in one location in the LLC for all queries and allocations from all cores. For each workload we present results for the highest performing baseline and apply our technique to it. We measure performance using the SimFlex multiprocessor sampling methodology [60].

In Section 6.4 we performed our circuit analysis targeting an industrial 65nm process (the latest process technology for which a design kit is available to us), in which a chip of this size is likely infeasible. However, we pursue these cache sizes and core microarchitecture to match the scale of our workloads and model a near-future server-class CMP. We configure our simulation with the cache latency and energy results derived in Section 6.4.

We study the TPC-C v3.0 OLTP workload on IBM DB2 v8 ESE and Oracle 10g Enterprise Database Server. We also evaluate a selection of SPLASH2 [61] and PARSEC 2.1 [6] benchmarks that are sensitive to on-chip access latency. These are *barnes*, *moldyn* and *ocean* from SPLASH2, and the *canneal* benchmark from the PARSEC 2.1. We found that the two OLTP workloads *db2* and *oracle*, as well as *barnes* and *moldyn*, favor the private organization. On the other hand, *ocean* and

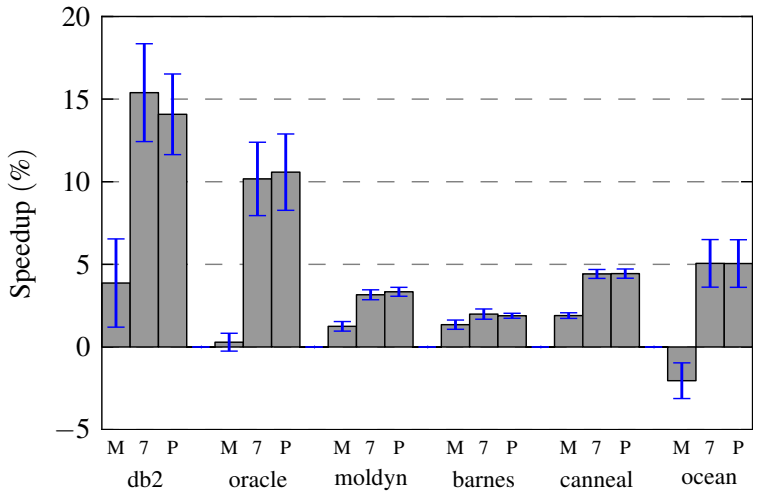



Figure 6.4: Percent speedup over sequential access.

*canneal* perform better on the shared baseline. The rest of this section presents these workloads running under their respective baselines.

### 6.5.2 Impact of Embedded Way Prediction

As shown in Figure 6.4, embedded way prediction with 7 partial tag bits (denoted 7) achieves the performance potential of parallel access (P). The figure shows speedup normalized to a sequential access baseline. The error bars indicate 95% confidence intervals obtained by our sampling methodology, thus the apparent speedup of our design with respect to parallel access is not statistically significant. Figure 6.5 shows a normalized breakdown of cycles-per-instruction spent on various stall sources for a range of designs. The graph is normalized to sequential access (S), and includes the same three designs as Figure 6.4 (including a breakdown for the sequential baseline). The time breakdowns are broken into (from bottom to top) busy time, stalls on store instructions, stalls on L1D accesses, stalls for L1I misses, stalls on data accesses to the LLC, stalls on main memory and other stall sources.

Though prior work [9, 38] has shown that predicting the MRU way (M) is effective in low-associativity L1 caches, it underperforms in the highly-associative LLC, achieving an average 17% of the speedup potential of parallel access.

We find the two database applications are particularly sensitive to our scheme because it accelerates the many L1I misses serviced at the LLC, with up to 15.4% speedup for *db2*. The instruction footprints of these two applications (over 2MB [20]) overwhelms the small L1I cache, creating a significant performance bottleneck. Unlike data stalls, these misses cannot usually be hidden through out-of-order execution,

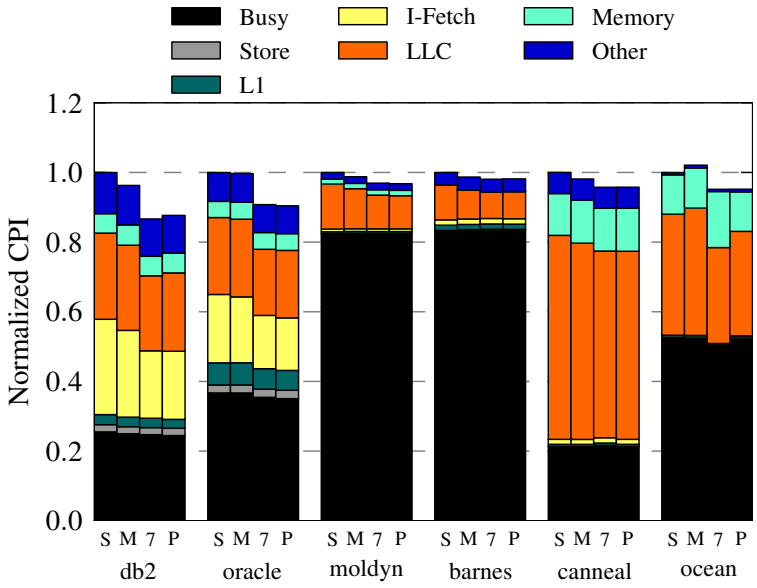



Figure 6.5: Impact on Cycles Per Instruction.

and prior work [15] indicates that adding an intermediate cache level (e.g., a 256KB L2) is not likely to be effective, as the instruction footprint is so large (over 2MB). Barnes and *moldyn* gain only modest benefits from parallel access, as their runtime is dominated by computation (busy time). Ocean is more sensitive to LLC time. Interestingly, *ocean* experiences a slowdown under MRU way prediction. MRU prediction is little better than random guessing for *ocean* (see next sub-section) and the additional data array bandwidth pressure created by mispredictions (each misprediction incurs two data array accesses) leads to significant queueing delays in bandwidth-intensive execution phases. Although *canneal* exhibits a large fraction of LLC stalls, these misses are largely coherence misses (i.e., accesses to dirty data), which spend their time traversing the on-chip network rather than in an LLC tile.

### 6.5.3 Sensitivity to Partial Tag Width

In Figure 6.6, we examine the sensitivity of prediction accuracy to the width of the partial tag comparison and explore the impact of the inhibit bit to reduce partial tag collisions. The prediction is successful if it activates the correct way during a cache hit, either because the partial tag match identifies a unique way (Predicted-Unique), or the inhibit bit correctly discerns amongst colliding partial tags (Predicted-Collision). When the cache access will miss, the predictor ideally should not activate an erroneous way (NoPredict-Miss). On the other hand, mispredictions occur when

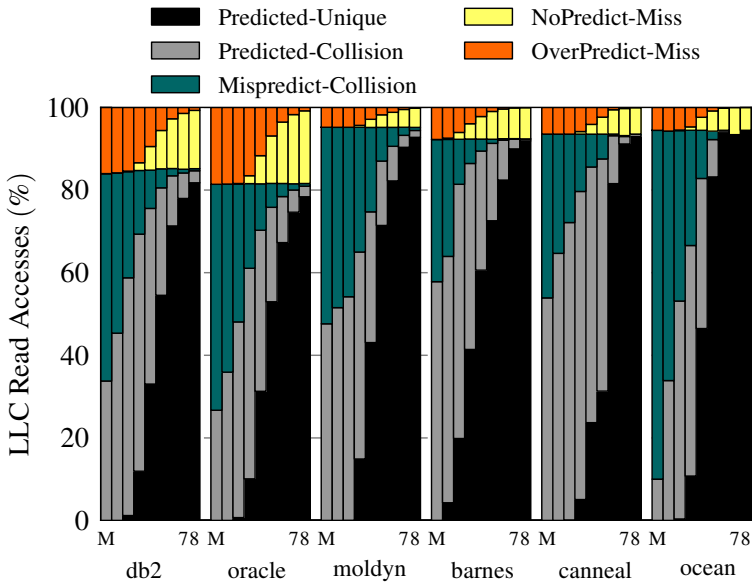


Figure 6.6: Impact of partial tag matching.

collisions obscure which way to predict during a hit (Mispredict-Collision) or when any way is predicted during a miss (OverPredict-Miss).

Naturally, as we increase partial tag width from 0 to 8 bits, the Predicted-Unique and NoPredict-Miss fractions increase steadily towards perfect accuracy. The relative size of the Predicted-Collision segment indicates the importance of the inhibit bit. Broadly, the results indicate that inhibit bits are critical for accurate prediction when the partial tag width is 4 or less, while their impact shrinks for wider partial tags. The inhibit bit plays no role during misses.

From these results, it is clear that server applications require 6-8 partial tag bits to maximize prediction accuracy, in contrast to the 3-4 partial tag bits recommended in earlier studies [42, 11, 63]. We also see that MRU-based prediction (M)—equivalent to a partial tag width of zero (0)—never achieves prediction accuracy over 65% and can never inhibit a data array access during a cache miss.

### 6.5.4 Power and Energy

We turn finally to examine the power and energy impacts of embedded way prediction. Figure 6.7 shows absolute LLC power, while Figure 6.8 shows normalized LLC energy per instruction, which is equivalent to an energy-delay product. Although embedded way prediction increases power by 13% on average over sequential access, we find that its performance benefits compensate for its power costs through savings in leakage energy, resulting in a net EPI increase of 11%. This energy efficiency improvement arises because the cache leakage power is amortized over more

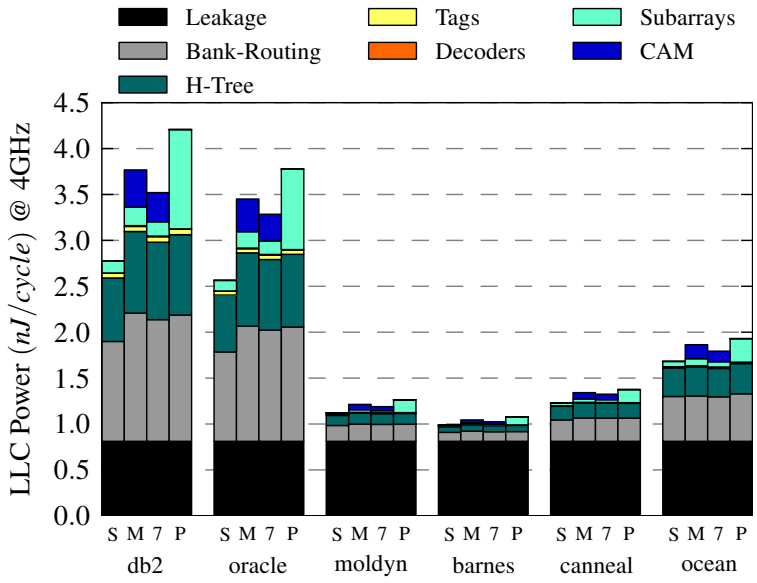



Figure 6.7: LLC power.

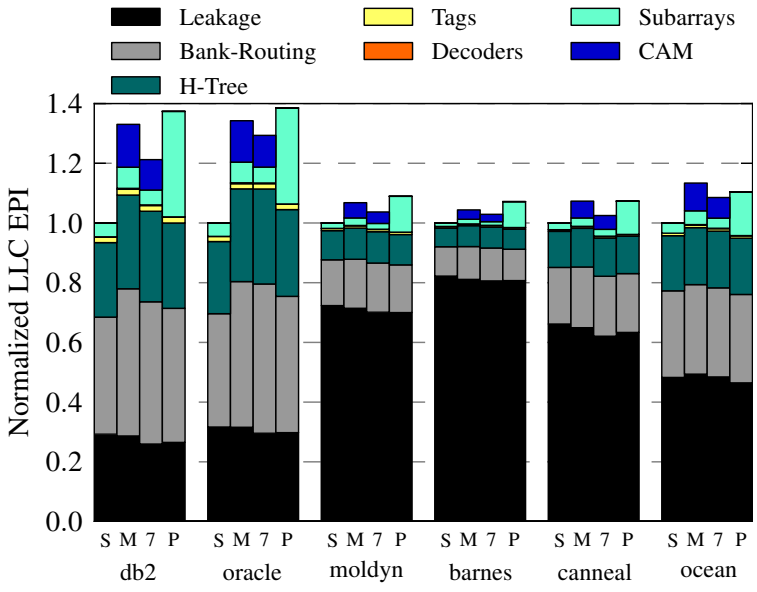



Figure 6.8: Normalized LLC energy per instruction.

instructions per unit time. Comparatively, parallel access incurs an almost doubled power increase of 23.4%, resulting in an EPI increase of 17.5%. The dynamic power overhead of parallel access is higher than the other designs, because it activates all 16-ways each access, whereas our design activates at most two. While a full-system power analysis is beyond the scope of this chapter, the EPI metric is expected to improve for high performing designs.

## 6.6 Summary

Server applications are growing increasingly sensitive to on-chip cache access latency, as larger capacities allow larger working sets to be captured on chip. In this chapter, we have revisited way-prediction using partial tag matching as a means to accelerate accesses in large LLCs without abandoning the energy efficiency advantages of sequential cache access. The central challenge of deploying way prediction in highly-associative LLCs is to enable the wider partial tag comparison called for in the server context while still overlapping the partial tag comparison with wordline decode. To this end, we have proposed embedded way prediction, an architecture and circuit design that embeds a dynamic CAM circuit within data sub-array decoders. We demonstrate that embedded way prediction achieves all the potential performance of parallel lookup, improving performance by up to 16% (7% on average) while incurring only a 8.2% average increase in LLC energy per instruction.

# CHAPTER VII

# Conclusion

High performance core and cache designs often utilize power-hungry techniques to parallelize instruction execution and memory accesses, resulting in unnecessary operations that waste energy. In contrast, energy-efficient designs take a minimalistic approach, opting to execute just the necessary operations in a serial fashion. While these designs apply their techniques to all operations in a one-size-fits-all approach, a judicious mix of parallel and serial execution has the potential to achieve the best of both high-performing and energy-efficient designs. This thesis examined hybrid designs for both the core and cache.

We proposed a novel, hybrid out-of-order/in-order core, which can opt for either of the two execution mechanisms at instruction-by-instruction granularity. We revisited way-prediction for L1 caches, a hybrid of parallel and sequential tag-data access, and evolve it for large, highly associative last-level caches. Both the cache and core demonstrate the potential for hybrid designs to offer the computer architect a more compelling performance-power trade-off.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12), Dec 2003.

[2] B.S. Amrutur and M.A. Horowitz. Fast low-power decoders for rams. *IEEE J. Solid State Circuits*, 2001.

[3] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proc. 30th Int'l Symposium on Computer Architecture (ISCA)*, Jun 2003.

[4] Amirali Baniasadi and Andreas Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proc. 33rd Int'l Symposium on Microarchitecture (MICRO)*, MICRO 33, Dec 2000.

[5] S. Bell et al. Tile64-processor: A 64-core SoC with mesh interconnect. In *IEEE Intnl. Solid-State Circuits Conf.*, 2008.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), Aug 2011.

[8] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proc. 11th Great Lakes Symposium on VLSI (GLSVLSI)*, Mar 2001.

[9] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. Intnl. Symp. on High-Performance Computer Architecture*, 1996.

[10] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, Hkan Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2), March 2009.

[11] Jian Chen, Ruihua Peng, and Yuzhuo Fu. Low power set-associative cache with single-cycle partial tag comparison. In *6th Intnl. Conf. on ASIC*, 2005.

[12] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proc. 25th Int'l Symposium on Computer Architecture (ISCA)*, Jun 1998.

[13] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *Proc. 10th Int'l Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2004.

[14] A El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in smt processors. In *Proc. 9th Int'l Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2003.

[15] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proc. 17th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[16] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Proc. 41st Ann. Intnl. Symp. on Microarchitecture*, 2008.

[17] Brian Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *Proc. 28th Int'l Symposium on Computer Architecture (ISCA)*, May 2001.

[18] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. 20th Int'l Conference on Parallel Processing (ICPP)*, Aug 1991.

[19] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *Proc. 4th Int'l Symposium on High Performance Computer Architecture (HPCA)*, Feb 1998.

[20] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer Architecture*, 2009.

[21] A. Hilton, S. Nagarakatte, and A. Roth. icfp: Tolerating all-level cache misses in in-order processors. In *Proc. 15th Int'l Symposium on High Performance Computer Architecture (HPCA)*, Feb 2009.

[22] A. Hilton and A. Roth. Bolt: Energy-efficient out-of-order latency-tolerant execution. In *Proc. 16th Int'l Symposium on High Performance Computer Architecture (HPCA)*, Jan 2010.

[23] S. Hily and A Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proc. 5th Int'l Symposium on High-Performance Computer Architecture (HPCA)*, Jan 1999.

[24] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proc. of the Intnl. Symp. on Low Power Electronics and Design*, 1999.

[25] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. 34th Int'l Symposium on Computer Architecture (ISCA)*, Jun 2007.

[26] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction. In *Proc. 37th Ann. Intnl. Symp. on Computer Architecture*, 2010.

[27] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proc. 17th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2012.

[28] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proc. 16th Ann. Intnl. Symp. on Computer Architecture*, 1989.

[29] K. Khubaib, M.A. Suleman, M. Hashemi, C. Wilkerson, and Y.N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proc. 45th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2012.

[30] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *Proc. 40th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2007.

[31] N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proc 35th Ann. Intnl. Symp on Microarchitecture*, 2002.

[32] H. Kodata, J. Miyake, Y. Nishimichi, H. Kudo, and K. Kagawa. An 8kb content-addressable and reentrant memory. In *Intnl. Solid-State Circuits Conf.*, 1985.

[33] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2), March 2003.

[34] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proc. 5th European Conference on Computer systems*, Apr 2010.

[35] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. 36th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2003.

[36] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. 42nd Int'l Symposium on Microarchitecture (MICRO)*, Dec 2009.

[37] Sam Likun Xi, Hans Jacobson, Pradi Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *Proc. 21st Int'l Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[38] Lishing Liu. Cache designs with partial address matching. In *Proc. Ann. Intnl. Symp. on Microarchitecture*, 1994.

[39] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. 45th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2012.

[40] José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. 35th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2002.

[41] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proc. 18th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2013.

[42] Rui Min, Zhiyong Xu, Yiming Hu, and Wen-ben Jone. Partial tag comparison: A new technology for power-efficient set-associative cache designs. In *Proc. 17th Intnl. Conf. on VLSI Design*, 2004.

[43] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Labs, April 2009.

[44] H.H. Najaf-abadi and E. Rotenberg. Architectural contesting. In *Proc. 15th Int'l Symposium on High Performance Computer Architecture (HPCA)*, Feb 2009.

[45] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. 24th Int'l Symposium on Computer Architecture (ISCA)*, May 1997.

[46] Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. A novel tag access scheme for low power l2 cache. In *Proc. Design, Automation Test in Europe Conf.*, 2011.

[47] Il Park, Chong Liang Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *Proc. 36th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2003.

[48] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. 34th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2001.

[49] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The metaflow architecture. *IEEE Micro*, 11(3), June 1991.

[50] M. Powell, A. Agrawal, TN Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc, 34th Ann. Intnl. Symp. on Microarchitecture*, 2001.

[51] Pierre Salverda and Craig Zilles. A criticality analysis of clustering in superscalar processors. In *Proc. 38th Int'l Symposium on Microarchitecture (MICRO)*, Nov 2005.

[52] Ryota Shioya, Masahiro Goshima, and Hideki Ando. A front-end execution architecture for high energy efficiency. In *Proc. 47th Int'l Symposium on Microarchitecture (MICRO)*, Dec 2014.

[53] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proc. 12th Int'l Symposium on Computer Architecture (ISCA)*, Jun 1985.

[54] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *Proc. 11th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2004.

[55] F. Tseng and Y.N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proc. 35th Int'l Symposium on Computer Architecture (ISCA)*, Jun 2008.

[56] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd Int'l Symposium on Computer Architecture (ISCA)*, May 1996.

[57] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd Int'l Symposium on Computer Architecture (ISCA)*, Jun 1995.

[58] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proc. 39th Int'l Symposium on Computer Architecture (ISCA)*, Jun 2012.

[59] Hans Vandierendonck, Philippe Manet, Thibault Delavallee, Igor Loiselle, and Jean-Didier Legat. By-passing the out-of-order execution pipeline to increase energy-efficiency. In *Proceedings of the 4th International Conference on Computing Frontiers (CF)*, May 2007.

[60] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26:18–31, 2006.

[61] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, 1995.

[62] Se-Hyun Yang and Babak Falsafi. Near-optimal precharging in high-performance nanoscale cmos caches. In *Proc. 36th Annual Intnl. Symp. on Microarchitecture*, 2003.

[63] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. *ACM Trans. on Architecture and Code Optimization*, 2(1), 2005.