EXPLORING AND EVALUATING

CONTROL-FLOW AND THREAD-LEVEL PARALLELISM

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chen-Yong Cher

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2004

To my parents.

ACKNOWLEDGEMENTS

I want to express my most sincere appreciation to my advisor, Dr. T. N. Vijaykumar, who has given me patient guidance and encouragement during my study. His unquenchable enthusiasm for research and excellence inspires me to reach for perfection in life.

I also want to thank my parents, Cher Yeong How and Ng Wei San, for their lifelong guidance, encouragement and unconditional love. I am thankful to my parents and my grandmother for the sacrifices they have made for me.

Many special thanks to all my family members and my friends. I am grateful to my brothers, Cher Chen Chiang and Cher Chen Lung, who generously offer their support and advice when I need them. I want to thank my "architecture" best friends, An Chow Lai, Il Park and Michael D. Powell, for offering genuine friendship and formidable challenges throughout my study. I also want to thank all the friends whom I have been priviledged to know over the years, for making my stay in West Lafayette a happy and unforgettable one.

I want to thank Purdue University for offering me the opportunity for education. I am grateful to the professors who generously shared their knowledge in many subjects and the staffs who kindly offer their help in many ways.

Finally, this dissertation is to the memory of my grandfather, Choo Tow Meng, whose love and encouragement continue to inspire me to this day.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Chen-Yong Cher. Ph.D., Purdue University, May, 2004. Exploring and Evaluating Control-flow and Thread-level Parallelism. Major Professor: T. N. Vijaykumar.

Simultaneous Multithreading (SMT) has been proposed for improving processor throughput by overlapping multiple threads in a wide-issue superscalar processor. While trace cache, value prediction, and prefetching are shown to be effective in improving single-thread performance in the superscalar, there has been no analysis of these techniques in an SMT processor.

SMT brings new factors both for and against these techniques, and it is not known how these techniques would fare in SMT. Therefore, I evaluate these techniques in an SMT. My key contributions are: (1) I identify a fundamental interaction between the techniques and SMT's sharing of resources among multiple threads, and (2) I quantify the impact of this interaction on SMT throughput. SMT's sharing of the instruction storage, physical registers, and issue queue impacts the effectiveness of trace cache, value prediction, and prefetching, respectively. My simulations show that (1) compared to a similar-sized i-cache, trace cache degrades throughput; (2) with a typical number of physical registers, value prediction degrades throughput; and (3) For memory-intensive workloads, prefetching improves throughput with many threads. For workloads with mixed memory demand, prefetching has little opportunity and slightly degrades throughput.

Although modern superscalar processors achieve high branch prediction accuracy, certain branches either are inherently difficult to predict or incur destructive interference in prediction tables, causing significant performance loss due to mispredictions. I propose a novel micro-architecture, called Skipper, to handle such difficult branches by exploiting control-flow independence. Skipper altogether avoids incorrect instructions by skipping

over, without even fetching, the control-flow dependent computation conditioned by a difficult branch. Instead, Skipper fetches and executes the control-flow independent instructions which need to be executed irrespective of the branch's outcome. Because Skipper executes the correct control-flow dependent instructions after the difficult branch is resolved, it conserves the valuable resources.

Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in a superscalar pipeline. Skipper fetches the skipped control-flow dependent instructions after the difficult branch is resolved, out of program order. SPECint95 simulations shows that Skipper performs 10% and 8% better than superscalar and previously proposed Polypath, respectively, when all three micro-architectures have equal i-cache bandwidth and hardware resources.

1 INTRODUCTION

Two architectures for mainstream computing today are Simultaneous Multithreading (SMT) and superscalar processors. The micro-architecture community has made impressive improvements in superscalar by introducing myriad novel techniques over the years. Because these techniques introduce little or no change to the semantic between processor and applications, they significantly improve performance while preserving compatibility for existing applications. Because SMT is designed by extending the capability of a superscalar core to handle multiple, independent threads, SMT naturally inherits these techniques that are originally intended for superscalar. However, the effectiveness of these techniques have not been studied in the context of SMT. Because SMT operates fundamentally different from a superscalar, the benefit of these techniques in an SMT is questionable.

SMT has been proposed for improving processor throughput by overlapping multiple threads in a wide-issue superscalar processor. Separately, three techniques which are used to improve single-thread performance by exploiting more instruction-level parallelism (ILP) in superscalar are: 1) trace cache to increase fetch bandwidth, 2) value prediction to break data dependences, and 3) prefetching to hide memory latency. While these techniques have evolved over the years by many researchers and have been shown to be effective in the single-threaded superscalar, there has been no analysis of their effectiveness in SMT, which is becoming the microarchitecture of choice for high-performance microprocessors (e.g., Intel's Hyperthreading, Sun's Niagara, IBM's POWER5).

Compared to superscalar, SMT brings new factors both for and against these techniques, and it is not known how these techniques would fare in SMT. An study of how these technique fare in SMT is critical to the design decision of whether to include them in

an SMT. On one hand, including an effective technique will improve SMT performance; on the other hand, including an ineffective technique will waste design time, chip area and power, and even hurt SMT performance. Because SMT are typically built by converting existing superscalar cores, some of these techniques that are already implemented in superscalars will also be included automatically in SMT. Because these techniques improve single-thread performance, future Operating Systems should enable the techniques all the time when running as a superscalar, but enable selectively depending on the effectiveness of these techniques when running as an SMT. In the environment where some threads are prioritized over the others, because these techniques may be more effective on certain threads regardless of their priority, the techniques may improve low priority threads while hurting higher priority threads. In such cases, the Operating System (OS) should selectively enable the techniques only for high priority threads. Traditionally, OSs are designed assuming that each thread uses one superscalar core, even in an Symmetric Multi-Processor system (SMP), and enforcing thread priority through time-slicing. Consequently, OSs need not disable hardware optimizations in such systems. My finding that recommends such OS intervention in an SMT is a significant departure from the traditional OS design. Therefore, it is important to know how the three techniques fare in SMT.

My study fills this important gap by evaluating these techniques in the context of an out-of-order issue SMT and provides recommendations for future SMT designs. Because SMT's goal is to improve throughput, which is an important performance metric for server-class machines which increasingly use SMT, I evaluate the techniques in terms of processor throughput. My novelty is not in the study or improvement of these techniques, but in their evaluation in the context of SMT. My key contributions are: (1) I identify a fundamental interaction between the techniques and SMT's sharing of resources among multiple threads, and (2) I quantify the impact of this interaction on SMT throughput. This interaction is the key issue and common theme in my evaluation of the three techniques.

Previous studies showed that trace cache increases fetch bandwidth [37,36,34,13,5]. Trace cache creates traces from dynamic instruction sequences and allows an entire trace to be fetched in one access. A key motivation for trace cache is that increasing fetch

bandwidth in superscalar is complicated and involves more than merely using many fetch ports. To utilize multiple fetch ports, superscalar needs multiple branch prediction, which is hard to implement. Implementing multiple branch prediction involves both (1) maintaining high accuracy of prediction and (2) providing multiple, contiguous fetch (Program Counter) PCs for the same thread. Trace cache handles these issues effectively and achieves better performance than multiple branch prediction.

Unfortunately, trace cache introduces multiple copies of instructions in different traces, despite the most efficient implementation [5]. This redundancy reduces the effective size of the cache. Increasing the cache size is difficult due to latency, area, and power considerations. This trade-off of space for bandwidth seems reasonable for superscalar because a single thread may not need a large instruction cache. However, SMT needs a larger instruction storage (i.e., trace cache or i-cache) because multiple threads *share the storage.* In contrast to superscalar, an SMT can supply multiple fetch PCs from different threads and utilize extra fetch ports effectively without needing multiple branch prediction. Therefore, it is not clear whether trace cache's trade-off of space for bandwidth will improve SMT throughput.

Value prediction predicts values instead of waiting for long-latency dependences to be resolved, speeding up computation even beyond data-flow limits [30,2,40,32,9]. Prediction accuracy can be increased and the benefit of the technique can be sustained by trading off coverage and predicting only highly-predictable long-latency operations (e.g., cache misses) [9]. In contrast to using value prediction in superscalar, SMT can simply tolerate L1 misses with execution of other threads. For L2 misses which have long latency, SMT squashes the thread [47], releasing the thread's shared resources (i.e., physical registers and issue queue slots) to allow overlapping of the L2 miss with other threads.

Applying value prediction to SMT raises a key but subtle issue related to *sharing of registers*: value prediction holds up physical registers even when the prediction is correct! Due to program-order commit, instructions that follow a correctly-value-predicted, long-latency instruction hold up registers even after completing execution. These instructions can release their registers *only after* the long-latency instruction completes, confirms the

prediction, and commits. Building larger register file to alleviate such hold-up proves hard due to latency, area, and power considerations [4,6,33]. While this hold-up of registers may be acceptable for superscalar because they would be unused otherwise, it may not be profitable for SMT, in which multiple threads create a higher demand for the shared registers. Therefore, it is not clear whether SMT throughput is improved more by value-predicting long-latency instructions and holding up registers; or by squashing the instructions and releasing registers so other threads can use the registers and overlap the latency.

Prefetching predicts future memory references and brings data into caches before the data is actually needed [44,26,45,10,18,25,55,16]. Recent proposals for aggressive hardware prefetching, such as Dead-block predictor[25] and its successor Time-Keeping predictor [55], are highly successful even with non-strided access patterns. SMT has two opposing effects on the opportunity available from prefetching. On one hand, because SMT can tolerate cache misses, it may present less opportunity to prefetching. On the other hand, because SMT issues memory references from multiple threads, it increases the pressure on the memory hierarchy and may present more opportunity.

Prefetching in SMT achieves coverage and accuracy comparable to those of a single thread. However, prefetching raises a subtle issue related to *sharing of the issue queue*. While prefetching into L2 achieves most of the benefit of prefetching into L1 without incurring L1's contention problems for a single thread [16], prefetching only into L2 causes a problem for SMT. Prefetching into L2 converts slow L2 misses into fast L2 hits; however, the L2 hits still miss in L1, resulting in the same L1 misses occurring in *fewer* cycles. L1 misses clog the issue queue with dependent instructions, even though L1 misses are short. While SMT without prefetching is also clogged for the L1-miss duration, it eventually incurs an L2 miss and squashes the thread [9], unclogging the issue queue to allow other threads to progress. Because prefetching causes L1 misses to occur in fewer cycles, the issue queue is clogged more often with prefetching than without. Thus, even correct prefetching may hurt SMT throughput! Unfortunately, neither removing L1 misses nor circumventing them to avoid the clogging is easy; removing L1 misses by prefetching into L1 is difficult due to L1's high contention [16], which is worse in SMT.

Table 1: Trade-offs of trace cache, value prediction and prefetching

	Trace Cache
Pros	- satisfies SMT's high demand of fetch bandwidth.
Cons	- causes redundancy in instruction storage while SMT demands high instruction storage capacity. - SMT provides high fetch bandwidth without needing multiple branch prediction.
	Value Prediction
Pros	- breaks the data dependences of individual threads.
Cons	- Data dependence delay can be hidden by other threads. - holds up resources even when predictions are correct.
	Prefetching
Pros	- fulfills SMT's high demand on memory access.
Cons	- Opportunity may drop because of thread-level parallelism. - may cause issue queue clogging even when prefetches are correct

Circumventing L1 misses is also difficult because L1 misses are known too late in the pipeline to prevent dependent instructions from entering the pipeline. Thus, in addition to the uncertainty in opportunity, the question of whether issue-queue clogging or latency hiding will impact SMT throughput more is unclear. Table 1 summarizes the trade-offs of the techniques when they are implemented in SMT.

In superscalar, the micro-architecture community has also made impressive improvements in branch prediction [29,53] to avoid pipeline stalls caused by branches, achieving prediction accuracies that are typically 95% or higher. Nevertheless, certain branches are either inherently hard to predict or incur destructive interference in a finite-sized branch prediction table. Mispredictions of such "difficult" branches cause considerable performance penalty and will worsen as the pipeline stages increase in the future microprocessors. One approach to target such difficult branches is to exploit the fundamental property of control-flow independence. Despite advances in out-of-order techniques to exploit data independence, the technique that exploits control-flow independence has not been extensively studied even in the context of a superscalar. To reduce branch misprediction penalty

on superscalar, I propose a novel architecture, called Skipper, to handle difficult branches. Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in the context of a superscalar pipeline.

Skipper avoids predicting difficult branches, by skipping over the computation conditioned by a difficult branch, and exploits the fundamental property of control-flow independence. The computations in a branch's taken and non-taken paths are conditioned by the branch, and are control-flow dependent on the branch, because whether each of the computations should be executed or not depends on the outcome of the branch. In contrast, the computations immediately following the point where the taken and non-taken paths re-converge are control-flow independent, because the *post-reconvergence* computations will be executed, irrespective of whether the branch is taken or not. A previous study shows potential speedups of about 30% in a wide-issue superscalar by exploiting control-flow independence [38].

Previous proposals to handle difficult branches are (1) to execute both the taken and non-taken paths conditioned by the difficult branch or (2) upon a mis-prediction, selectively recovering control-flow independent instructions by not squashing data-independent instructions, and re-executing only instructions that are data-dependent or (3) compiler-driven techniques, such as delay slot and conditional execution. Because the first approach executes both paths, one of which is incorrect, and the second execute incorrect instructions from the mis-predicted path, both approaches squandering processor cycles and valuable resources such as i-cache bandwidth. Incorrect instructions are numerous because they include not only incorrect control-flow dependent instructions but also control-flow independent instructions which are data dependent on the incorrect control- flow dependent instructions. In [38], the proponents of the second approach conclude that the biggest performance limiter is wasted resources consumed by incorrect control dependent instructions. The third approach executes incorrect instructions or waste resources, even when the branches can be correctly predicted without the compiler's help. Delay slot does not work well in long pipeline and wastes resources when the compiler cannot find independent instructions at compile time and resulting in empty delay slots; conditional execution executes both taken and non-taken paths and discards computations from the incorrect path, when the branch is resolved.

To conserve valuable resources, Skipper altogether avoids incorrect instructions by skipping over, without even fetching, the control-flow dependent instructions from both taken and non-taken paths conditioned by a difficult branch. Skipper fetches and executes instructions from post-reconvergent instructions when waiting for a branch to resolve, and executes only the correct control-flow dependent instructions after the difficult branch is resolved. Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in the context of a superscalar pipeline. Superscalar employs sophisticated out-of-order instruction-issue techniques which routinely skip over data dependent instructions but not control-flow dependent instructions. Other approaches, employing hardware and software assists vastly different from a superscalar, skip over instructions to pursue multiple flows of control: Multiscalar[42] uses the compiler to identify reconvergence points, Dynamic Multithreading [1] uses hardware to skip over loops and calls, but not branches.

Unlike superscalars which always fetch instructions in predicted program order, Skipper fetches the skipped control-flow dependent instructions after the post-convergent instructions, out of program order. Thereby, Skipper exploits control-flow independence of the post-reconvergent computation, and overlaps execution of computation before the branch (and resolution of the difficult branch) with the execution of the post-reconvergent computation. Execution overlaps comes from post-reconvergent instructions that are data dependent of the skipped instructions. Skipper forces the post-reconvergent instructions, which are data dependent on the yet-to-be-fetched skipped computation, to wait till the difficult branch is resolved and the correct path within the skipped computation is fetched and executed. Note that conventional superscalars delay all instructions following a mispredicted branch till the instructions are re-executed. In contrast, Skipper delays only the skipped instructions and the post-reconvergent data-dependent instructions, but does not delay the post-reconvergent data-independent instructions.

I describe four mechanisms to implement Skipper in an out-of-order pipeline: First, to identify difficult branches, Skipper uses the previously proposed JRS scheme. Second, to determine difficult branches' reconvergence points, Skipper employs a heuristic based on

idiomatic control-flow code patterns generated by modern compilers for conditional constructs, without requiring scanning of instructions as in [38]. Third, despite out-of-order fetching of the skipped instructions, Skipper maintains program order in the instruction window and load/store queue. On fetching a difficult branch, Skipper creates a appropriate sized, contiguous gap in the instruction window and load/store queue, to be filled later by the skipped computation from the correct path Fourth, to force data-dependent, post-reconvergence instructions to wait till the yet-to-be-fetched skipped instructions execute, Skipper estimates register dependencies, learning from prior dynamic instances. At the time of skipping, Skipper updates the register rename tables using this dependence information, making post-reconvergent data-dependent instructions wait.

The main contributions of this dissertation are:

- I show for the first time the evaluation of trace cache, value prediction and prefetching in the context of an Simultaneous Multithreading Processor (SMT).
- My evaluation pin-points when-and-why these techniques do or do not benefit SMT, thereby providing recommendations for future SMT designs. Because my results show that trace cache and value prediction hurt SMT throughput, I recommend that these techniques to be excluded for future SMT designs when multi-programmed workloads are the common case and throughput is the main goal.
- My findings also create a new responsibility for the OS: Because the techniques improve single-thread performance, I recommend that the OS disable the techniques when running multi-programmed workload, and enable them for single-threaded workload and for high-priority threads in a multi-programmed workload.
- I propose Skipper, the first proposal to skip control- flow dependent instructions, without wasting resources on incorrect control-flow dependent instructions.
- I describe key mechanisms to implement Skipper without unduly complicating the pipeline despite out-of-order fetching.

For the evaluation of trace cache, value prediction and prefetching in SMT, I use multi-programmed workloads that comprises a subset of the SPEC2000 benchmarks. The main results of my simulations are:

- Trace cache, value prediction and prefetching significantly improve single-thread performance. This result agrees with previous papers and validates my implementations.
- Given similar size for the duo of trace cache and backup i-cache as the conventional i-cache, trace cache degrades SMT throughput compared to the conventional i-cache. Throughput improves for 2 threads, agreeing with the two-threaded Pentium IV's use of a trace cache. This result shows that trace cache's space-for-bandwidth trade-off hurts SMT. Giving considerable extra size to the trace cache results in the trace cache performing only marginally better than the conventional i-cache, showing that trace cache is not effective in SMT.
- Given a typical number of physical registers, value prediction degrades SMT throughput, showing that holding up registers under value prediction hurts SMT throughput. While value prediction does improve individual threads that have long-latency misses, it does so at the cost of the other threads, defeating SMT's purpose. Using infinite physical registers and perfect confidence prediction results in value prediction performing only marginally better than conventional SMT, showing that value prediction is not effective in SMT.
- For memory-intensive workloads, there is substantial opportunity for prefetching even with many threads, showing that SMT cannot hide not all long-latency misses. I found that prefetch coverage can be reduced to balance prefetching and issue queue clogging, improving throughput for this workload. For workloads with mixed memory demand, SMT significantly reduces opportunity. Despite reducing the coverage, prefetching slightly degrades throughput for this workload due to issue queue clogging. Like value prediction, prefetching also improves individual threads at the cost of the other threads in this workload, degrading overall throughput.
- My simulations using SPECint95 show that Skipper performs 10% and 8% better than superscalar and the previously proposed Polypath, respectively, when the three architectures have equal i-cache bandwidth and hardware resources.

Section 4 gives the background of trace cache, value prediction, prefetching and control-flow independence. Section 3 describes the details of Skipper microarchitecture. Section 4 describes the methodology. Section 5 shows my results, and Section 6 concludes my dissertation.

2 BACKGROUND

Figure 1 shows how the trace cache, value prediction and prefetching affect the superscalar pipeline.

2.1 Trace Cache

Before trace cache was introduced, Tyson et al. [54] increased fetch bandwidth by predicting multiple branches every cycle with Branch Address Cache. Rotenberg et al. [37] introduced trace cache, and compared it with other high-bandwidth instruction fetch schemes. Others [36,35,13] studied important issues concerning trace cache performance such as partial matching, cache associativity, fill unit, and multiple branch prediction. Patel et al. [34] proposed branch promotion and trace packing for improving trace cache bandwidth. To achieve better utilization of trace cache space, Black et al. [5] suggested the block-based trace cache, which stores pointers to blocks constituting a trace, instead of storing instructions. Any repetition of the traces results in only the pointers being repeated instead of the entire trace, reducing space requirements.

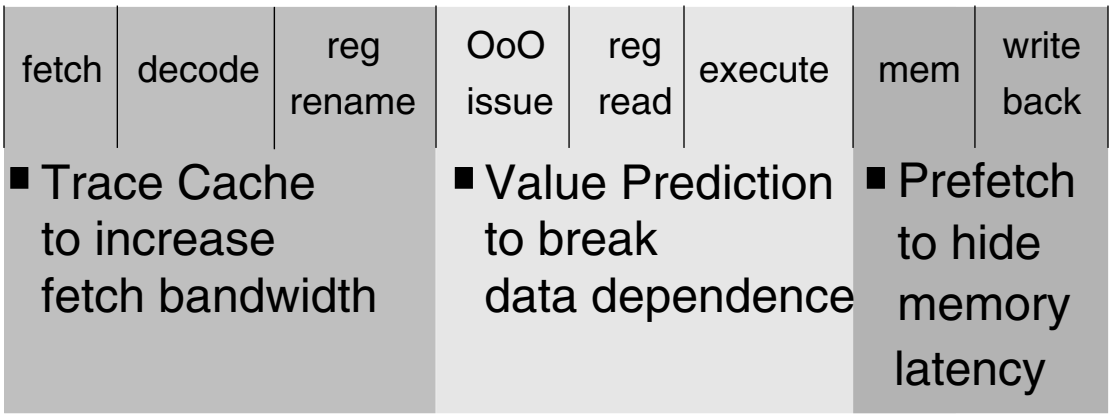


Figure 1: Trace cache, value prediction and prefetching in superscalar pipeline

Because I wish to study the effectiveness of trace cache's space-for-bandwidth trade-off, and because the block-based trace cache is the most space-efficient implementation that also achieves high bandwidth, I use the block-based trace cache in my evaluations. I discuss the details of this specific trace cache later in Section 4.1.

2.2 Value Prediction

Lipasti et al. [30] proposed last-value prediction with saturating confidence counters. Mendelson et al. [2] added a stride prediction scheme, and Farkas et al. [40] studied the implementation details for the context-based prediction scheme. Others predict load values by using recent store information [32,50]. However, without accurate prediction, value prediction may hurt performance due to misprediction penalties unless there is hardware support, such as selective recovery, to reduce the penalty. Calder et al. [9] showed the importance of confidence prediction to perform selective value prediction to avoid mispredictions and achieve good speedup even without the complicated machinery of selective recovery.

In addition to avoiding complicated selective recovery, reducing mispredictions is important for SMT so that processor resources are not wasted on incorrect execution. Therefore, I use [9]'s selective value prediction, which combines confidence prediction and value prediction, in my evaluations and discuss their details in Section 4.2.

2.3 Prefetching

While prefetching can be implemented in either software [44,26] or hardware, I focus on hardware prefetching in this study. Chen et al. [45] proposed the stride prefetcher, and others [19,12] used a stream buffer for prefetching. Markov prefetching uses address correlation (i.e., correlation among addresses in the cache miss stream) to improve the accuracy of prefetching arbitrary, non-strided access patterns [10,18]. These proposals focused on *what* to prefetch but do not pinpoint *when* to prefetch.

Lai et al. [25] first proposed to consider the timing of memory access patterns to determine when to prefetch and improves accuracy over [18]. They introduced Dead-Block Predictors to predict the dead blocks — i.e., the blocks that will be evicted without any use

— in L1. When a block dies, the prefetcher predicts the next access to the block's set and prefetches the next access into the dead block's frame. Kaxiras et al. [20] also proposed a scheme to predict dead blocks, but they used the prediction for reducing cache leakage power and not for prefetching data. Hu et al. [55] applied [20] to prefetching and used smaller prediction tables than [25] for both dead-block prediction and next-address prediction while achieving better performance.

Lastly, Hu, et al. [16] simplified [55] by showing that when prefetching into a large highly-associative L2 cache, dead-block prediction was not necessary. [16] also showed that prefetching into L2 can achieve most of the benefit of prefetching into L1 without disrupting the highly contentious L1 with untimely or incorrect prefetches. Because SMT's multiple threads cause even higher contention on L1 than that of superscalar, I implement the latest, best-performing tag-correlating prefetching of [16] to prefetch into L2 without disrupting L1. I discuss the prefetcher implementation details in Section 4.3.

2.4 Control-flow Independence

There have been several results on the potential of exploiting control-flow independence [38]. Many previous ideas to handle difficult branches, amount to executing both the taken and not-taken paths, using varying degrees of ISA support for predication. Proposals such as Multipath [52], Polypath [22,23], dual pipelines [24] and instruction windows [11], and Dynamic Hammock Predication [21] explicitly follows this approach. ISA support for predicated execution removes difficult branches, but at the cost of executing instructions from both the taken and not-taken paths [28,3].

Researchers [38] have proposed selective recovery of control-flow independent instructions after a misprediction, but they point out in a later paper that the scheme is hard to implement [39]. Selective squashing may require expanding/contracting the instruction window at multiple, arbitrary points because the incorrect and correct path instructions are intertwined in the instruction window. Out-of-order pipelines usually track data dependencies through register rename map tables at the granularity of a block of instructions (typically between successive branches), and not individual instructions. This coarse granularity reduces the number of map tables and makes misprediction handling fast and

efficient, but disallows fast extraction of selective information about individual instructions. For a realistic selective recovery [39], they propose using Trace processors' hierarchical organization, a solution not applicable to superscalars (Although Pentium IV has a trace cache, it is not a trace processor). Instruction Reuse [41] is a general technique which can recover values of control-flow independent instructions after a misprediction. But instruction reuse also squashes all instructions following a misprediction. Multiscalar [42] and Dynamic Mutithreading [1] uses hardware or compiler to demarcate threads, which may choose control-flow independent threads to shield intra-thread mispredictions from squashing other threads.

3 SKIPPER MICROARCHITECTURE

In this section, I discuss how Skipper is mapped to a superscalar microarchitecture at a high level.

Figure 2 illustrates the differences between correct prediction, misprediction, and skipping. Figure 2 (a) identifies the control-flow dependent (segments A and B) and control-flow independent, post-reconvergent (segment C) computations in a program segment, as defined in Section 1. Figure 2 (b) shows the time lines of correct and incorrect predictions. Correct prediction leads to execution overlap among the instructions before the branch, and the instructions from the predicted path (A and C segments). A misprediction usually leads to squashing of all instructions after the branch, irrespective of whether they are control-flow dependent or independent (both B and C segments).

Figure 2 (b) shows Skipper's time line. Skipper overlaps the computation before the difficult branch (and resolution of the branch), with the post-reconvergent instructions that are data independent of the skipped instructions (data independent instructions from segment C). On resolving the difficult branch, Skipper suspends execution of the post-reconvergent instructions. Skipper then executes the correct path in the skipped computation (segment A), allowing the post-reconvergent instructions that are data dependent on the skipped instructions to proceed (rest of segment C). After fetching all the skipped instructions till the reconvergence point, Skipper continues with the suspended post-reconvergent computation.

Despite its advantages, skipping is not always beneficial. Skipping branches that would be correctly predicted may cause performance loss, while not skipping branches that would be incorrectly predicted results in lost opportunity. Comparing correct prediction and skipped time lines in Figure 2 (b) reveals this point. The performance loss is

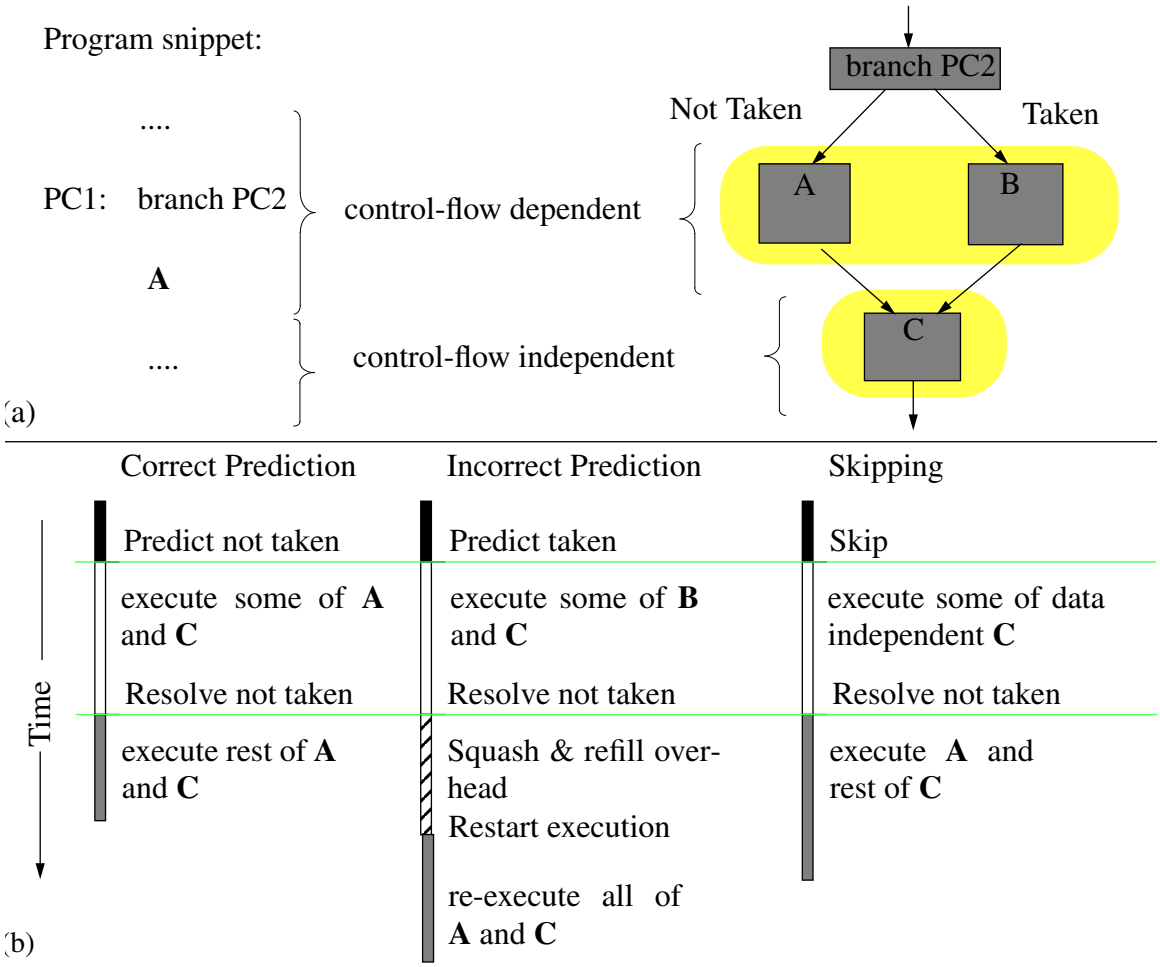


Figure 2: Exploiting control-flow independence.

incurred because conventional superscalars do not delay any of the instructions following a correctly predicted branch, but Skipper unnecessarily delays the skipped instructions and the post-reconvergent data-dependent instructions (from C), until the difficult branch is resolved.

3.1 Overview

I describe Skipper in the context of an out-of-order pipeline that uses rename tables for register renaming and an instruction window for out-of-order issue. Skipper employs JRS scheme [17] to identify branches that are repeatedly mispredicted. Basically JRS monitors the prediction accuracy of prior instances of branches and isolates branches with low accuracy. If a branch is identified as hard-to-predict, Skipper uses a heuristic to determine the branch's reconvergence PC, and maintains the PC in the Skipped Computation Informa-

tion Table (SCIT). Subsequent instances of the branch obtain the reconvergence PC from the SCIT.

In the following cycles, Skipper fetches the post-reconvergence instructions and places them in the instruction window. Skipper creates a contiguous gap, large enough to hold all the skipped instructions, in both taken and non-taken paths, in the instruction window, and places the post-reconvergence instructions after the gap, similar to [43]. Skipper learns the likely maximum gap length by counting number of instructions in skipped computation, from both taken and non-taken paths, in prior execution of the branch. After the skipped branch resolves, Skipper fetches skipped instructions in the correct path (taken or not taken), and places them in the gap. Instructions in the instruction window remain in program order, and Skipper maintains precise interrupts despite out-of-order fetching.

3.1.1 Maintaining Data Dependence

Using the fact that conventional superscalar fetches instructions in program order, register rename tables links register producers to consumers, and the load/store queue deduces producer-consumer relationships for memory values. Because Skipper fetches instructions out-of-order, the post-reconvergence instructions before the skipped instructions, the rename tables cannot establish correct data dependencies among the skipped instructions and the post-reconvergence instructions. Previous schemes that fetch instructions out of order face similar problems: The Multiscalar architecture uses the compiler to specify register dependencies [7]. The Dynamic Multithreading architecture employs value speculation and intricate recovery [1].

In conventional out-of-order pipelines' rename stage, instructions map their architecture destination register to a new physical register, and place the new architectural-to-physical mapping in the master rename table. Out-of-order fetch presents two issues for Skipper's register renaming. First, Skipper has to ensure that rename maps for the skipped instructions' source register are not clobbered by the post-reconvergent instructions. Second, Skipper has to ensure that the post-reconvergent consumer instructions obtain the correct rename maps corresponding to the skipped producer instructions. To handle these

issues, Skipper learns the set of architectural source registers, the *inputreg set*, and destination registers, the *outputreg set*, for the skipped instructions (both taken and not taken) in prior dynamic instances. The SCIT holds the inputreg and outputreg sets. The outputreg set is similar to Multiscalar's create mask, except Multiscalar uses the compiler to determine this information [51], and Skipper uses hardware.

For the first issue, Skipper copies the master rename table to a backup rename table, at the time of skipping a branch. To conserve bandwidth, Skipper only copies rename table entries that correspond to the union of inputreg set and outputreg set. At this point, the master table reflects the register state of the program at the difficult branch. Post-reconvergent instructions modify the master table, and not the backup table. Later, when the skipped instructions are fetched, they use the maps in the backup table.

For the second issue, Skipper forces the data-dependent, post-reconvergence instructions to wait until the yet-to-be-fetched skipped instructions execute. The outputreg set gives Skipper a priori, albeit approximate, knowledge of the destination registers for which the yet-to-be-fetched skipped instructions. At the time of skipping, Skipper *preallocates* and *preassigns* physical register, one for each architectural register in outputreg set, and marks the physical registers busy. Much like a data-dependent instruction in out-of-order superscalars, all post-reconvergent instructions that are possibly data-dependent on a skipped instruction waits until the corresponding physical register is ready or bypassed. When the skipped instruction eventually completes execution, its preassigned physical register gets the value, allowing all waiting post-reconvergent instructions to proceed. Data-independent, post-reconvergence instructions proceed without waiting, much as in superscalars. Because several skipped computations could be in-flight, Skipper uses multiple backup and preassign tables, much as superscalars use a backup rename table for each unresolved branch in flight.

For memory dependencies, Skipper faces the same problem of maintaining program order in the load/store queue, as in the instruction window. Skipper creates an appropriately-sized gap in the load/store queue in parallel with the instruction window, and maintains the load/store queue gap length information also in the SCIT. Skipper uses

conventional load/store queue's ability to allow loads to proceed without knowing all previous store addresses, letting post-reconvergent loads to proceed even though the skipped stores have not even been fetched. Conventional load/store queues check if later loads complete prematurely before an earlier store to the same address, and enforce store-load program order via squash and rollback. Skipper can avoid such squashes using well-known memory dependence synchronization techniques [31]. Thus, Skipper's loads and stores remain in program order and in correct data dependencies in the load/store queue, despite out-of-order fetch.

3.2 Supporting control-flow independence in Skipper

Before I describe the details of how the required information is gathered in SCIT (Skipped Computation Information Table), I explain how I use the SCIT information.

The Skipper pipeline treats instructions that are not skipped as well as branches that are not difficult much like a conventional superscalar pipeline. The cases where Skipper's action are different from those of a conventional pipelines are (1) when Skipper identifies a branch to be difficult, (2) when Skipper resolves a difficult branch, (3) when Skipper fetches and executes a skipped computation, and (4) when Skipper fetches the last instructions from a skipped computation. The post-reconvergence instructions flow through the pipeline without any special actions. Figure 3 shows an out-of-order pipeline extended with Skipper. I do now show post-reconvergence instructions.

3.2.1 Fetching a difficult branch

Using the predicted PC, the front-end of the Skipper pipeline probes the JRS structure and the SCIT, in addition to the usual branch prediction tables. If the JRS structure identifies a branch to be difficult, the fetch stage fetches from the reconvergence PC provided by the SCIT. If the SCIT does not have an entry for this branch or if the instruction window gap length as provided by the SCIT entry is larger than the gap-length-threshold, then Skipper defaults to branch prediction, overruling the JRS's recommendation. Gap-length-threshold ensures that Skipper does not create inordinately large gaps in the instruction window, under utilizing the instruction window. Skipper obtains the reconvergence PC in

parallel with the fetching of the difficult branch, much like a branch target address from the BTB in conventional pipelines, Thus, Skipper fetches the post-reconvergent instructions in the immediately following cycle after the difficult branch fetch cycle, without inserting any bubbles in the pipeline

Skipper allocates an entry in the Skipped Instruction Status Table (SIST) for every skipped branch to hold information required by various pipeline stages for the skipped computation and the post-reconvergent computation. There could be multiple difficult branches in flight in the pipeline, and the SIST holds an entry for every difficult branch in flight. However, Skipper does not perform nested skipping (i.e., skipping within a skipped computation), and so the SIST entries are in program order. Every instruction carries its SIST entry number so that the instruction can be associated with its SIST entry in later stages of the pipeline.

Conventional pipelines allocate a history rename table when a branch enters the rename stage. Subsequent instructions copy the previous rename map of their destination registers from the master table to the history table, before placing their new rename map in the master table. This procedure essentially checkpoints the rename maps, thus allowing fast recovery of the maps on mispredictions. The copying is done at a rate matching the issue width. For instance, in a four-issue machine, the rename table allows, in one cycle, eight reads for the sources, and four reads and eight writes to checkpoint the old maps and update the new maps for the destinations. Rename table bandwidth is a critical resource, and as such the entire table (e.g., for 64 architectural registers and 512 physical registers, the table has 576 bits) cannot be backed-up en masse, in one cycle. In [52], this point was noted in the Mapping Synchronization Bus description.

When the difficult branch reaches the rename stage, Skipper modifies the rename tables as per the inputreg and outputreg sets. To that end, Skipper first allocates a backup rename table and copies the rename maps for the skipped instructions' inputreg and outputreg sets into the backup table. In Section 3.1, I explained why the backup table holds the inputreg maps, but the reason for copying the outputreg maps will become clear in

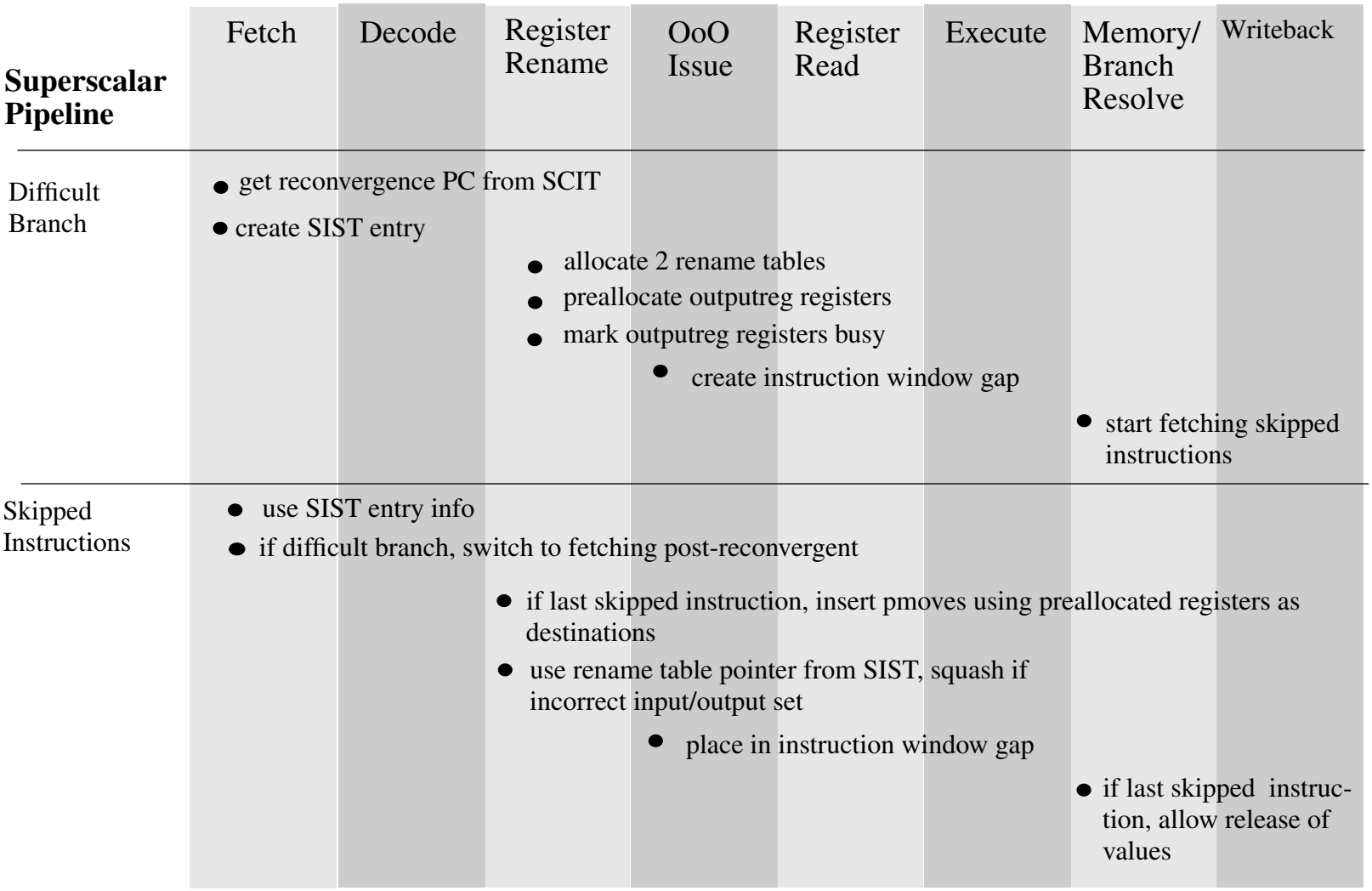


Figure 3: Skipper pipeline

Section 3.2.3. This copying proceeds at the bandwidth provided by the backup table and may stall the rename stage for as many cycles as needed for the copying.

Skipper then preallocates and preassigns new physical registers to the skipped instructions' outputreg set. Skipper updates the master table with the preassigned physical register rename maps, and marks the preassigned physical registers as busy. Additionally, Skipper allocates another preassign rename table and updates the table with the outputreg set's preassigned physical register rename maps. The updating of the master and preassign tables too proceed at the bandwidth provided. Skipper places pointers to the backup and preassign tables in the difficult branch's SIST entry so that when the skipped instructions are fetched, the pipeline knows which rename table to use.

In the out-of-order issue (OoO issue) stage, Skipper uses the difficult branch's instruction window and load/store queue gap length information from the SCIT to create a gap in the instruction window and load/store queue. Skipper puts pointers to the instruction window and load/store queue gaps in the difficult branch's SIST entry so that on fetching the skipped instructions, the pipeline knows where to place them.

3.2.2 Resolving a difficult branch

Till Skipper resolves the difficult branch, execution proceeds with the post-reconvergent instructions much like conventional pipelines, and the pipeline front-end predicts branches. The post-reconvergent instructions modify the master table, as usual. If JRS identifies a subsequent branch to be difficult, Skipper continues at the branch's reconvergence points, allowing multiple skipped branches in flight.

Upon resolving a difficult branch, the fetch stage is diverted to fetch from the correct path of the skipped computation, temporarily suspending fetching from the post-reconvergent computation. Skipper provides the correct branch target to the fetch stage along with the branch's SIST entry number so that the skipped instructions are associated with the correct SIST entry. Skipper holds the PC up to which the post-reconvergent instructions have been fetched in the difficult branch's SIST entry, so that after the skipped instructions are all fetched, the fetch stage can revert back to fetching the post-reconvergent

instructions starting from that PC. The skipped instructions from the correct path enter the pipeline starting from the cycle following the branch resolution. This change of fetch stream does not entail any pipeline bubbles because the post-reconvergent instructions flow through the pipeline, as before.

If the post-reconvergent instructions fill up the instruction window (except for the gap) and the front-end pipeline stages from OoO issue all the way back to fetch, Skipper may deadlock. Basically the skipped instructions cannot get into the pipeline even though there are instruction window slots set aside for them. Skipper avoids such deadlocks by squashing the instructions in the stages from OoO issue back to fetch, freeing up the front-end stages so that the skipped instructions can get into the instruction window.

3.2.3 Fetching and executing skipped instructions

The skipped instructions, carrying the SIST entry numbers provided by the difficult branch, pass through the pipeline. Skipper places the skipped instructions in the instruction window, and loads and stores in the load/store queue using the SIST entry's instruction window entry pointer and load/store queue entry pointer, respectively. The skipped instructions use the rename tables identified by the backup and preassign table pointers, stored in the SIST entry. The backup table contains both inputreg and outputreg registers' rename maps corresponding to the register state of the program at the difficult branch.

The skipped instructions use the backup (and not master) table both to get their source rename maps and to put their destination rename maps. If a skipped instruction's source is an inputreg register, the backup table provides the rename map for the register. There are two issues with guaranteeing correctness with regard to the outputreg registers. First, multiple skipped instructions writing to the same architectural destination register pose a problem because Skipper preassigns only one physical register per outputreg register. Second, Skipper needs to identify when it is safe for the dependent post-reconvergent instructions waiting on the preassigned registers to use the values in the registers. Because multiple skipped instructions may write to the same architectural register, it may not be correct to allow a dependent instruction to read the register as soon as a write occurs.

Skipper handles both issues using a simple approach. In the rename stage, the skipped instructions do not use the preassigned physical registers as their destinations. Instead, these instructions obtain newly allocated physical registers. As skipped instructions pass through the rename stage, they update the backup table with the new physical register maps. Subsequent skipped instructions obtain the correct rename maps for their source registers from the backup table. At the end of the skipped computation, Skipper introduces extra physical register move instructions called pmoves, similar to [21,42]. Pmoves (described in Section 3.2.4) copy the latest outputreg value from the physical registers given by the backup table maps to the preassigned registers given by the preassign table maps.

Because the outputreg set is an estimate based on previous instances, an outputreg register may not be written by the skipped instructions. In that case, the latest value for an outputreg register comes from an instruction before the difficult branch. It is for this reason Skipper copies the outputreg rename maps into the backup table when the difficult branch is in the rename stage, as mentioned in Section 3.2.1. Consequently, the backup table holds the latest rename map for the outputreg registers irrespective of whether the skipped instructions actually write to the outputreg registers or not, and therefore, the pmoves copy the correct values.

Conventional pipelines free the previous physical register mapped to the same architectural register as the committing instruction's destination. Because Skipper commits instructions in program order, this approach also works for Skipper. Clearly this approach works for all instructions up to the first gap. At the gap, previous physical registers fall into two categories: either they are mapped to outputreg registers or not. Those mapped to outputreg registers are freed by writes in the gap, and the writes' registers are freed by pmoves; if there are no writes, pmoves directly free the previous registers. Those not mapped to outputreg registers are freed by post-gap instructions, as usual.

It is possible that an architecture register not in the outputreg set is written to in the skipped computation. A dependent post-reconvergent instruction may incorrectly use a stale value assuming that the register would not be written by the skipped instructions. A

similar situation is possible for the inputreg set, where a skipped instruction needs to read a register not in the inputreg set. These conditions are easily detected in the register rename stage by comparing each skipped instruction's destination (source) register against the outputreg (inputreg) set of the instruction's SIST entry. On detection, Skipper simply squashes all post-reconvergent instructions and triggers recovery of the missing register's rename map, irrespective of whether or not an incorrect value or rename map was used.

While executing the skipped instructions, Skipper predicts the branches within the skipped computation, as usual. Incorrect branch prediction within the skipped computation results in squashing all post-reconvergent instructions, nullifying Skipper's ability to exploit control-flow independence. If JRS identifies a branch within skipped computation to be difficult, Skipper suspends fetching from the branch till the branch is resolved and reverts to fetching from the post-reconvergent stream, using the post reconvergent fetch PC in the SIST entry. While this simple solution further delays the dependent post-reconvergent instructions, it avoids squashing post-reconvergent computation. Another solution is to skip the difficult branches within the skipped computation but such nested skipping may complicate implementation.

Out-of-order fetching may interact with branch prediction unfavorably because speculative update of branch history [15] may be disrupted by the out-of-order fetch stream. Because this is the first research on this approach, I avoid this issue by assuming that branch prediction updates occur at commit point, although previous results have shown speculative updates to perform better than commit updates.

3.2.4 Last instruction in the skipped computation

Each SIST entry holds the corresponding reconvergence PC to allow the fetch stage to determine when a skipped instruction stream merges with its post-reconvergent computation and stop fetching more instructions from the skipped stream. Every cycle, the fetch stage compares the next fetch PC with the reconvergence PCs held in the SIST entries, and on a match stops fetching from the corresponding skipped computation further. Skipper then inserts the extra pmoves into the instruction window, so that they execute as and when

the value for the outputreg registers become available. On execution, the pmoves write to the preassigned physical registers and mark them ready, allowing dependent, post-reconvergent instructions to proceed. The fetch stage then reverts to the post-reconvergent computation by continuing from the PC at which the post-reconvergent stream was left off.

Because the instruction window and load/store queue gap lengths are estimates based on previous instances, it is possible that the gaps in the instruction window and load/ store queue fill up before all the skipped instructions are fetched. In that case, Skipper simply squashes all the post-reconvergent instructions to make room for the rest of the skipped instructions to be placed in the instruction window. If the reconvergence PC obtained by the pattern-matching heuristic is incorrect, the effect of this incorrect information is that the instruction window gap fills up before the skipped instruction stream merges with the post-reconvergent computation, causing Skipper to squash all the post-reconvergent instructions starting from the incorrect reconvergence PC.

3.3 Learning the SCIT information

Skipper learns all the required information about the skipped computation from previous instances and deposits them in the SCIT for subsequent instances. The information collected in the SCIT are: identifying which branches are difficult, what the reconvergence PC are, what the instruction window and load/store gap lengths should be, and the skipped instructions' inputreg/outputreg set.

3.3.1 JRS for Identifying the difficult branches to skip

Skipper uses JRS branch confidence estimation, previously proposed by [17], to identify difficult branches by accessing the JRS structure with every branch prediction. JRS tracks number of times a branch has been mis-predicted using saturating counters much like a branch predictor. The counters count up on incorrect predictions and count down on correct predictions. JRS chooses appropriate values for both the up/down rates, and the count threshold above which a branch is deemed difficult. Even if a branch is deemed difficult, JRS and branch continue to make predictions and update the tables. If a branch is repeatedly predicted correctly, JRS stops marking the branch as difficult.

The key aspect of JRS relevant to Skipper, is that skipping branches that would be correctly predicted may cause performance loss, while not skipping branches that would be incorrectly predicted results in lost opportunity. The performance loss is incurred because Skipper unnecessarily delays the skipped instructions and data-dependent instructions located after the post-reconvergence point. Thus, there is a trade-off between JRS's coverage and accuracy, and while lower coverage means lost opportunity; lower accuracy may mean performance loss.

3.3.2 Heuristics for identifying reconvergence point

For if-then-else constructs in high-level languages, the compiler typically generates a branch to determine whether the if clause or the else clause is to be executed. The compiler also generates a jump instruction to the reconvergence PC at the end of the if clause, to elide the else clause. Therefore, the reconvergence PC can be determined if the jump instruction is located. The target PC of the branch is the beginning of the else clause, and the jump instructions is located immediately before the branch target. Skipper computes the target of the difficult branch and uses the PC immediately before the branch target to probe the i-cache and inspects the instruction there. If the instruction is a jump instruction, then the target of the jump is the reconvergence PC.

If the instruction at the PC immediately before the difficult branch target is not a jump instruction, then Skipper assumes that the difficult branch is from an if-then construct, instead of an if-then-else construct. For if-then construct, the branch target is the reconvergence PC. If a difficult branch is a backward branch (indicated by a negative offset), then Skipper assumes that difficult branch is a loop branch, which the loop iterations is hard to predict, and assigns the immediate PC after the branch as reconvergence PC. Unlike previous work [38], I do not include the return PC of a function as the reconvergence PC of all branches within the function body because of gap-length-threshold constraints.

Using only one probe into i-cache, Skipper's heuristic determines the reconvergence PC. Because the probe is done only for difficult branches and not for all branches, and that too only if the SCIT does not have the reconvergence PC, this probe does not degrade

i-cache bandwidth. Once the reconvergence PC is recorded in the SCIT, the heuristic is not used until the SCIT replaces the branch's entry due to conflict or capacity issues. Because Skipper obtains the reconvergence PC from SCIT most of the time and not from the heuristic, computing the difficult branch's reconvergence PC can be slow. Consequently, this computation is done over many pipeline stages instead of just decode, without affecting the cycle time.

There are compiler optimizations that may confuse the heuristic. For instance, in code layout optimization to improve i-cache performance, the compiler moves infrequent control-paths away from the sequential stream. Such code motion changes the code pattern and renders the heuristic ineffective. However, this optimization may be applied to only those branches that are biased towards one of the two paths; otherwise one path would not be more frequent than the other. Therefore, such branches may not be difficult to predict and may not need to be skipped. Other optimizations may cause exceeding of the gap-length-threshold. An example is tail duplication of the post-reconvergence code into the if and else path, which increases the gap length.

3.3.3 Estimating the gap length

After JRS identifies a difficult branch and the heuristic determines the reconvergence PC, Skipper collects the instruction window and load/store queue gap length information from subsequent instances of the branch. Upon committing the difficult branch, Skipper creates a valid entry in the Gap Information Learning Buffer (GILB), and places the reconvergence PC in the GILB entry. From the difficult branch onwards, every committing instruction increments the instruction window gap length count of *all* valid GILB entries, because each valid GILB entry represents a distinct difficult branch whose reconvergence PC has not been committed. Also, Skipper matches the PC of the committing instruction against the reconvergence PCs of all the valid GILB entries. A match indicates that the corresponding difficult branch's reconvergence PC has been reached. Skipper transfers the information in the GILB entries to the SCIT, and relinquishes the GILB entry.

Program Sniplet:

PC	Instruction
A	Branch B (difficult branch)
	If Clause
B-4	Jump C (C is the target of the jump instruction before B).
B	
	Else Clause
C	.... (C is the Reconvergence Point of branch A).

a) If-then-else construct

Program Sniplet:

PC	Instruction
A	Branch B (difficult branch)
	If Clause
B-4	
B	.... (B is the Reconvergence Point of branch A).

b) If construct

Figure 4: Examples of Skipper's Heuristic to determine reconvergent PC.

To keep the SCIT information as accurate as possible, Skipper continues to collect the information in subsequent instances of the difficult branch, irrespective of whether the branch is predicted or skipped. If the instruction window (or load/store queue) gap length count in any later instance is larger than the length recorded in the SCIT, Skipper updates the SCIT entry with the larger count. If the count is smaller, it is discarded. This repeated updating of the maximum length helps Skipper account for different control-flow path lengths within the skipped computation.

If Skipper does not maintain the maximum gap length, Skipper would essentially have to predict the skipped computation's path length to estimate the gap length. Predicting the path length may indirectly lead to predicting the difficult branch, defeating Skipper's purpose. Because the maximum gap length is longer than all but the longest path within the skipped computation, Skipper is conservative in setting aside the instruction window gap. Some of the instruction window slots remain empty if the actual path within the skipped computation is not the longest. This conservative choice is better than predicting the difficult branch because the number of wasted slots is still much smaller than the slots spent on numerous incorrect instructions in out-of-order superscalars and other approaches.

3.3.4 Determining inputreg and outputreg set

Along with the gap lengths, the GILB also tracks the outputreg sets and inputreg set of the skipped computation using a bit-vector field in the GILB entries. From the difficult branch onward, every committing instruction's destination (source) register is added to the outputreg (inputreg) bit-vector of all valid GILB entries. Because Skipper collects outputreg and inputreg information at commit point, incorrect predictions within the skipped computation do not adversely affect the accuracy of the information. When the gap length information is transferred to the SCIT from the GILB, the GILB entry's bit-vectors are OR'ed with the SCIT entry's corresponding bit-vector. The OR'ing accounts for difference outputreg and inputreg sets along different control-flow paths within skipped computation. The outputreg and inputreg sets are conservative union over all control-flow paths within the skipped computation. Not considering all the paths would cause Skipper to predicting the difficult branch indirectly, as argues above.

4 METHODOLOGY

I modified simplescalar [8], an execution-driven, cycle accurate simulator for my evaluation of trace cache, value prediction and prefetching in the context of SMT. My simulator carefully models SMT pipeline details, including out-of-order issue, memory-bus occupancy, multiple contexts, virtual-to-physical address translation, per-thread load/store queues and active lists, and shared physical register file and issue queue. The simulator models a pipeline that supports thread-level squashing on branch misprediction. To improve instruction throughput in SMT, I apply squashing on L2 misses [47], except for special cases that I will mention later. Because the L1 caches are virtually indexed, I use address offsetting described in [27] to spread out accesses of different threads in the cache. I also use the Bin-Hop page allocation policy to spread out accesses in the L2 cache [27].

My simulator runs *Alpha* binaries that are compiled with *peak* setting. I fast-forward the first two billion instructions of each thread. The fast-forwarding warms up branch predictor, the L2 and L1 caches, but do not gather statistics. I then simulate until one of the threads reaches 100 million instructions. For four or eight threads this method simulates more than 100 million instructions. Therefore, my results are unlikely to be biased by individual programs. Recently [46] proposes clustering phases to reduce simulation time while minimizing errors for simulating single program. However, clustering for a multi-programmed workload is more complicated and involves mixing phases of several programs. Because [46] does not show clustering for SMT simulations, I do not use such approach.

Multi-programmed workload is one of the most important workloads for SMT. To simulate real-world workloads, I choose sixteen benchmarks from SPEC2000 to compose

Table 2: Applications and multi-programmed workloads for SMT evaluation

Category	Benchmarks		
1T.ILP	**mesa**, **crafty, fma3d**, **eon,** **facerec**, **equake, sixtrack**, **galgel**		
1T.MEM	*vpr*, *apsi*, *art*, *applu*, *swim*, *lucas*, *mcf*, *ammp*		
1T.MIX	1T.ILP + 1T.MEM		
Workload	**Composition**	**Workload**	**Composition**
2T.ILP.1	**mesa, crafty**	2T.MIX.1	*vpr*, **mesa**
2T.ILP.2	**fma3d, eon**	2T.MIX.2	*apsi*, **crafty**
2T.ILP.3	**facerec, equake**	2T.MIX.3	*art*, **fma3d**
2T.ILP.4	**sixtrack, galgel**	2T.MIX.4	*applu*, **eon**
2T.MEM.1	*vpr, apsi*	2T.MIX.5	*swim*, **facerec**
2T.MEM.2	*art, applu*	2T.MIX.6	*lucas*, **equake**
2T.MEM.3	*swim, lucas*	2T.MIX.7	*mcf*, **sixtrack**
2T.MEM.4	*mcf, ammp*	2T.MIX.8	*ammp*, **galgel**
4T.ILP.1	2T.ILP.{1,2}	4T.MIX.1	2T.MIX.{1,2}
4T.ILP.2	2T.ILP.{3,4}	4T.MIX.2	2T.MIX.{3,4}
4T.MEM.1	2T.MEM.{1,2}	4T.MIX.3	2T.MIX.{5,6}
4T.MEM.2	2T.MEM.{3,4}	4T.MIX.4	2T.MIX.{7,8}
8T.ILP.1	4T.ILP.{1,2}	8T.MIX.1	4T.MIX.{1,2}
8T.MEM.1	4T.MEM.{1,2}	8T.MIX.2	4T.MIX.{3,4}

workloads that have two, four and eight threads. Out of these sixteen benchmarks, eight achieve the highest IPCs (shown in bold) and the other eight have the most L2 cache misses per instruction (shown in italics). I mixed these benchmarks to create three representative workloads of different ILP and memory demand. Table 2 lists the SPEC2000 benchmarks and multi-programmed workloads I use in this study. The first set, called *ILP,* consists of the high-ILP programs; the second set, called *MEM,* consists of the high-miss-rate programs; and the third set, called *MIX,* combines programs from both ILP and MEM. Within a set, there are four groups (*1 thread, 2 threads, 4 threads and 8 threads*) and each group indicates the workloads for a given number of threads. I use the ref input for all benchmarks.

Table 3 lists the configuration for the basic SMT in my study. I carefully choose an aggressive SMT core such that my results are representative of many different SMT configurations in the foreseeable future; a less aggressive SMT would handicap the techniques

Table 3: Base SMT configurations

Issue Width	8
L1 I-cache	64KB, 2-way, pipelined
L1 D-cache	64KB, 4-way, 3-cycle hit latency
L2 Cache	4MB, 8-way, 15-cycle hit latency
Memory	150 cycle latency, 4-cycle pipelined, split-transaction bus
Branch Predictor	16k/16k/16k spec-update, 8-cycle misprediction penalty
Physical Registers	100+*T**32 INT , 100+*T**32 FP for *T* threads
Active List	128/context
Load-Store Queue	64/context
Issue Queue	32 INT, 32 FP
MSHR	32

I study because of less headroom for improvements. I use an issue width of eight as other SMT-related previous studies do [49,47,48], unless otherwise specified. For branch prediction, I use a hybrid of local and *gshare* predictors. Each context uses a 128-entry return address stack and maintains its own branch history for the gshare predictor. The SMT in this study has two fetch ports and fills up fetch bandwidth from up to two threads. I use *ICOUNT* as my SMT fetch policy as recommended in [48].

I will describe the implementation details of trace cache, value prediction and prefetching and Skipper, in Section 4.1, Section 4.2, and Section 4.3, respectively.

4.1 Trace Cache Implementation

I implement the latest, most space-efficient block-based trace cache (TC) described in [5]. The TC is implemented using a block cache and a trace table. Each block of a block cache stores a small subtrace (e.g., a few consecutive basic blocks up to six instructions) and the trace table stores pointers to the block cache. To provide high bandwidth, the block cache is multi-ported (implemented via true ports and/or copies). The trace table provides *n* pointers which are used to pull out *n* subtraces from the block cache, and the subtraces together form the fetch unit of one trace. The subtraces are formed by observing past

instances of the instruction stream. The trace table is updated with pointers to the subtraces. Because the subtraces are small, there is less repetition than trace cache using full-blown traces [37,35,36,13,34]. Furthermore, only the pointers to subtraces, but not subtraces themselves, are repeated in the full traces, achieving further compaction.

Because my results show that TC is ineffective for SMT, I make the following assumptions to ensure that my results are not due to insufficient resources or inefficient implementation: 1) My TC uses an ideal, sequential, atomic multiple-branch predictor that accurately updates branch history even for branches predicted in the same cycle. In contrast, the base case SMT's i-cache uses a conventional, speculatively-updated predictor which predicts up to one taken branch or up to two branches per thread. The TC uses infinite branch-prediction bandwidth, therefore the branch promotion optimization in [34] is irrelevant. 2) The TC uses perfect target prediction for direct branches. 3) The TC has zero-cycle fill latency.

I implement the following key optimizations from [5]: 1) For termination, a subtrace ends upon encountering a branch, a jump, a call or a return instruction near the end of the subtrace. 2) My TC employs partial matching which allows a substring of a trace, instead of restricting to complete traces, to be supplied. 3) I use a two-way associative "rename table" to map PCs to trace pointers. The table determines whether a trace is present in the block cache on every TC access and handles replacement in the block cache. The table's associativity effectively makes each copy of block cache two-way associative. 4) On fetching, the processor sends a request to both TC and i-cache simultaneously. If the request misses in the TC but hits in the i-cache, there is a one-cycle penalty, as in [37,35,36,34,13,5]. 5) To compensate for block-level fragmentation, the TC provides more instructions than the processor's front-end width. The front end picks the number of instructions requested to send into the pipeline and buffers any excess instructions to be combined with the next trace. My TC has a six-instruction block size, as recommended in [5]. 6) I update the block cache speculatively on misses, as opposed to updating at commit. Other simulations (not shown) reveal that speculative update performs better.

When using the TC in SMT, I do the following to ensure that my SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) I employ address offsetting in the TC and its accompanying i-cache. 2) Each cycle, two threads access the TC and each thread gets half the TC bandwidth. My simulations (not shown) reveal that this policy achieves better performance than giving the full TC's bandwidth to only one thread.

4.2 Value Prediction Implementation

I implement the latest, best-performing selective value predictor (VP) described in [9]. The value predictor uses a confidence predictor to select when to predict and a hybrid of stride and context predictors to predict values. Because my results show that VP is ineffective for SMT, I make the following assumption to ensure that my results are not due to inefficient implementation: I assume that VP's value history is updated correctly by an oracle in the decode stage.

I implement the following key optimizations described in [9]: 1) To minimize mispredictions, I implement a history-based confidence predictor. 2) I employ warm-up counters so that instructions with insufficient history do not update predictors. 3) To reduce mispredictions and maximize the benefit, I allow the predicted value to be used only for load instructions that incur L1 misses. According to my evaluations, this scheme has better performance than one that predicts all instructions. While I use the predictions only on misses, I predict and update on all loads regardless of a hit or a miss to accelerate the predictor's warm-up. 4) Instructions that directly or indirectly consume predicted values are assigned lower priority and can execute only on otherwise-idle execution units. These instructions resume their normal priority when the prediction outcome is known. When a misprediction is detected, the pipeline squashes the thread's instructions that are subsequent to the producer. To avoid unnecessary squashing, squash does not happen if the mispredicted value has not been consumed.

When using VP in SMT, I do the following to ensure that my SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) Because VP benefits mostly from L2 misses, SMT's squashing on L2 misses would nullify much of the benefit of VP. Therefore, I modify the squashing policy on L2 misses in SMT. If an L2-missing load is

value-predicted, I do not squash the pipeline. This mechanism allows dependent instructions to consume predicted values and later release issue queue entries. When a thread fills up its active list or load/store queue on an value-predicted L2 miss, I squash the thread's instructions only in the front end and stall fetching from the thread until the miss returns. Otherwise, fetched instructions from the thread would clog the front end preventing other threads from making progress. This squashing of the front end is not extra because the base case already squashes the pipeline on all L2 misses, regardless of whether resources fill up. 2) To reduce aliasing in prediction tables, I add tags to all prediction table entries. 3) To avoid inter-thread interference in the prediction tables, I use per-thread prediction tables.

4.3 Prefetching Implementation

I implement the latest, best-performing tag-correlating prefetching (PF) [16]. The prefetcher decides what to prefetch by using the L1 miss stream as history to predict the next miss. The predictor is a two-level scheme, where the first level stores per-set miss stream history, and the second level stores the tag of next-misses. Upon an L1 miss, the prefetcher triggers prefetch to the predicted next miss. Instead of using dead-time prediction to trigger prefetches, this simplified prefetcher uses L1 misses as the triggers.

Because my results show that PF is effective for SMT, I do not give any undue advantage to the prefetcher to ensure that my results are not due to unjustifiable implementation assumptions. Specifically, because PF uses additional space for prediction tables, I compensate the base case by running it with a larger L2. Because the largest predictor size I use is 498KB (in an eight-thread SMT), I use a 4.5 MB L2 for all base case runs, while using only a 4MB L2 for all the PF runs. I enlarge the base case's L2 with no penalty to the L2 hit latency.

I implement the following key optimizations in the two-level predictor: 1) The predictor uses eleven bits of the L1 tag and one bit of the L1 index from the previous three misses, together with the full L1 tag from the previous miss, to form indexes into the second-level table as in [16]. 2) The first level uses per-set history as recommended in [55]. 3) The prefetcher uses 32 extra MSHRs to hold in-flight prefetch status and a 128-entry prefetch queue to hold pending prefetch requests, as recommended in [55].

When using PF in SMT, I do the following to ensure that my SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) While [55] prefetches data into L1, [16] argues that prefetching into L1 is difficult due to L1 contention. [16] shows that prefetching data only into L2 achieves most of the benefit of prefetching into L1 while entirely eliminating dead-block prediction. This effect is seen because L1 miss latency can be overlapped easily with ILP. While prefetching into L2 in SMT introduces the issue queue clogging described in Section 1, I could reduce prefetch coverage to balance prefetching and issue queue clogging and improve overall throughput. Therefore, I evaluate prefetching into L2. 2) The second-level table is accessed with the previous L1 miss and history, that is also made of L1 misses. Because the L1 is physically tagged, the L1 miss stream has physical addresses which are already randomized by bin-hopping (Section 4). Consequently, the second-level table does not need any offsetting to reduce conflicts. 3) Each level of prediction tables may be configured to be shared across threads or to be private to each thread. Because there is not much difference between shared or private for the second level, I use a shared second-level table. I show both private and shared configurations for the first-level table. 4) I increase the second-level shared table size with the number of threads (the table size is T * 120KB for T threads, except for eight-thread SMT I use T=4 to keep the size under 512KB). I increase the size with no change in the associativity, keeping the implementation reasonable. Although the table is shared among threads, no major conflicts among the threads occur because the table is accessed using physical addresses, which are unique across the threads.

4.4 Skipper Implementation

For Skipper, I modified the Simplescalar simulator [8] to model Skipper. Table 4 presents the SPECint95 benchmarks and their inputs used in this study. I run the benchmarks to completion. I assume a hybrid predictor and 9-cycle misprediction penalty. Table 5 shows the base system configuration parameters used throughout the experiments for Skipper, unless I specify otherwise. I assume generous branch prediction tables each of which has 8K entries to allow as high prediction accuracy as possible, but a modest SCIT size of about 3KB. I use a bimodal JRS with 4K, 4-bit entries for a total of 2KB. I model

Table 4. Skipper experimental benchmarks and parameters.

Benchmark	Input	Number of Instructions
cc1	cccp.i(test)	1.3 Billions
go	2stone9	548 Millions
li	test.lsp	957 Millions
perl	jumble	2.4 Billions
compress	train	36 Millions
ijpeg	vigo	1.5 Billions
m88ksim	ctl.in(test)	490 Millions
vortex	train	2.5 Billions

the return address stack (RAS), and account for RAS and BTB mispredictions which are not addressed by Skipper. I accurately model the extra rename bandwidth to handle inputreg and outputreg sets, and the extra pmoves at the end of skipped computation. I do not include any memory dependence synchronization mechanisms, but account for memory dependence squashes. Because Skipper's key advantage is in conserving i-cache bandwidth, I carefully model i-cache bandwidth.

Table 5. Skipper's base configurations.

Component	**Description**
Processor	8-way out-of-order issue,128-entry reorder buffer, 43-entry load/store queue
Branch prediction	hybrid 8K-entry bimodal, 8K-entry gshare, 8K 2-bit selector (total about 6KB) 128-entry RAS, 4-way 4K BTB (9-cycle misprediction penalty)
JRS	4K 4-bit bimodal, with threshold of 15(total 2KB); gap-length-threshold 48
L1 I-cache	64KB, 32-byte blocks, 2-way, pipelined, 2-cycle hit, lock-up free
L1 D-cache	64KB, 64-byte blocks, 2-way, pipelined, 2-cycle hit, lock-up free
L2 unified cache	2 Mbytes, 64-byte blocks, 8-way, 12-cycle hit, pipelined
Memory Bus	Split transaction, 32 bytes per bus cycle transfer
Main memory	Infinite capacity, 100cycle latency
SCIT	128 entries each 199-bit wide , 4-way (total size about 3KB): (21-bit tag, 32-bit reconvergence PC, 67-bit outputreg set mask, 67-bit inputreg set mask, 6-bit reorder buffer gap length, 6-bit load/store queue gap length)
SIST	36 entries each 192-bit wide (total about 864 bytes): (reorder buffer pointer 7 bits, load/store queue pointer 7 bits, fetch PC so far 32 bits, RAS pointer 7 bits, rename table pointer 5 bits, 67-bit outputreg set mask)
GILB	36 entries each similar to a SCIT entry (total about 896 bytes)

5 RESULTS

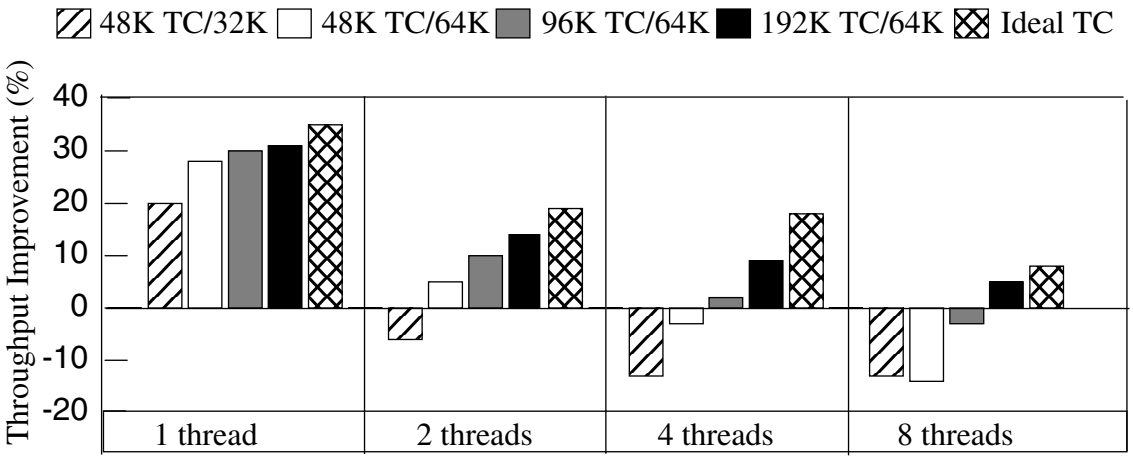
I present my results for trace cache, value prediction, and prefetching in Section 5.1, Section 5.2, and Section 5.3, respectively. As stated in Section 1, because SMT's goal is to improve throughput, which is also an important performance metric for server-class machines which increasingly use SMT, I evaluate these techniques in terms of instruction throughput. I present my results for the Skipper architecture in Section 5.4.

I find that (1) given similar size for the duo of trace cache and backup i-cache as the conventional i-cache, trace cache degrades SMT throughput compared to the conventional i-cache; (2) given a typical number of physical registers, value prediction degrades throughput; (3) prefetching improves throughput for memory-intensive workloads but degrades throughput for workloads with mixed memory demand. (4) Skipper performs 10% and 8% better than superscalar and Polypath for SPECint95, respectively, when the three architectures have equal i-cache bandwidth and hardware resources.

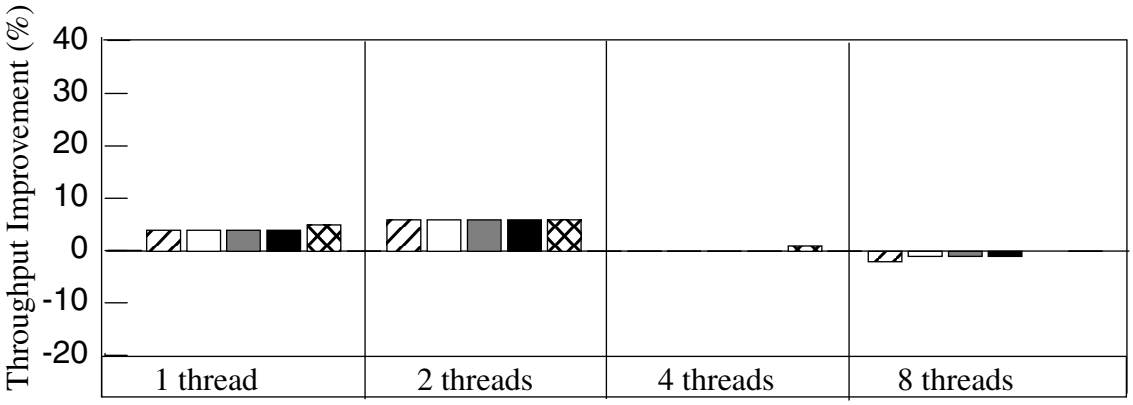
5.1 Trace Cache Results

Recall from Section 1 that TC trades off space for bandwidth and that the sharing of instruction storage among SMT's threads impacts this trade-off. In this section, I evaluate this trade-off in SMT. Because I found that TC benefits little with an 8-issue pipeline even for single-thread workloads (not shown), I use 16-issue width for the TC as in [37,35,36,34,13,5]. Accordingly, I also double the pipeline resources listed in Table 3, such as rename registers, issue queue, active list, load-store queue, and execution units.

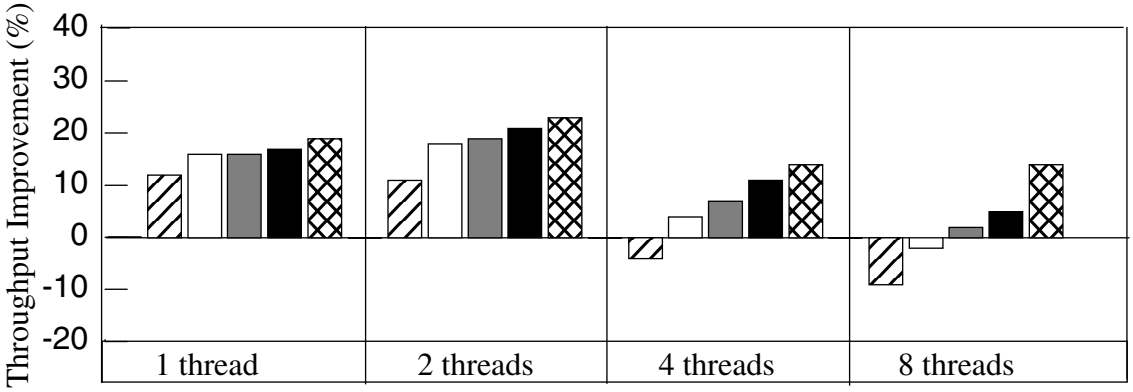
Figure 5 shows the throughput improvements of TC over the base case, which has 64KB i-cache and no TC. I show three sets of workloads: ILP, MEM, and MIX, as defined in Section 4. For each set, there are four groups of bars (*1 thread, 2 threads, 4 threads and*



(a) ILP Workload.



(b) MEM Workload.



(c) MIX Workload.

Figure 5: Trace cache throughput improvements.

8 threads) varying the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set.

Because I are interested in TC's space-for-bandwidth trade-off, I vary TC size. Within each group of bars for a given number of threads, the first bar shows a 48K TC using two copies of dual-ported block cache backed up by a dual-ported 32K i-cache, for a total size of 80K, compared to the base case of a dual-ported 64K i-cache. Thus, the first bar represents my comparison using a similar total size. The next three bars from left to right show a 64K i-cache combined with TC of 48K, 96K, and 192K (1-cycle latency). These bars represent the cases where the TC configurations use extra space compared to the base case of a 64K i-cache.

To examine the upper limit of improvement through enhancing TC, by increasing size/ associativity or using per-thread TC, I simulate *Ideal TC* which is an oracle configuration that has infinite size and always supplies as many instructions as the fetch bandwidth from two threads every cycle. Ideal TC does not suffer from fetch fragmentation or capacity/ conflict miss, and subsumes enhancements. Ideal TC uses the same ideal multiple branch predictor as other TCs mentioned in Section 4.1. The last bar shows Ideal TC's throughput improvement.

Figure 5 shows two clear trends. First, TC benefits ILP and MIX but not MEM. While ILP and MIX have enough parallelism that fetch bandwidth is important for performance, MEM is dominated by data cache misses that fetch bandwidth is not important. Second, for similar-size configurations, TC offers no benefit to SMT, and can lead to throughput degradation as the number of threads increases. This similar-size comparison is important because increasing the size of the level-1 instruction storage is difficult due to latency, area, and power considerations, as mention in Section 1. When I add an extra TC to the base 64K i-cache, TC is effective for single threads. This result agrees with previous papers [37,35,36,34,13,5] ([5] also gives extra space to TC), indicating that my TC implementation is correct. For two threads, TC improves SMT throughput; this results agrees with the two-threaded Pentium IV's use of a trace cache. However, when threads increase to more than two, TC's advantage rapidly diminishes. The *base case throughput*, shown in

the first row of Table 6, continues to improve as I increase the number of threads to eight, showing that TC's diminishing returns are *not* due to pipeline saturation. Even with a large, 192K TC with single-cycle latency, TC shows only modest improvement over the base case for four or more threads. These results are no surprise when we look at the last bar, which shows the throughput improvement with an ideal TC. The last bar clearly shows that TC's potential drops rapidly as thread increases. Thus, we see that SMT's sharing of instruction storage makes TC's space-for-bandwidth trade-off unprofitable.

In SMT, applying a technique may impact low-IPC threads and high-IPC threads differently. With the goal of maximizing throughput, SMT distributes resources (fetch and front-end bandwidth, issue queue slots, etc.) among threads proportional to each thread's individual IPC (e.g., using ICOUNT). However, applying a technique may improve a low-IPC thread, fooling SMT into allocating more resources to the improved-but-still-low-IPC thread at the cost of other high-IPC threads, reducing overall throughput. That is, one thread may improve but the overall throughput may reduce. To capture such cases, [47] introduces *weighted speedup*, which is the geometric mean of IPC improvements of each thread. If the weighted speedup is more than throughput improvement, then the technique impacts (positively or negatively) low-IPC threads more than high-IPC threads; if the weighted speedup is less than throughput improvement, then the reverse is true. If the two metrics are similar, then the impact on low- and high-IPC threads are similar. Although my goal is to evaluate processor throughput, I show weighted speedup for 48K TC with 64K i-cache in the second row in Table 6. We see that weighted speedup follows the same trend as throughput, confirming that TC's advantage diminishes as threads increase.

To explain TC's downward trend with an increasing number of threads, I compare base case i-cache miss rates with TC miss rates. The third row in Table 6 shows the i-cache *miss rate* in the base case, and the fourth and fifth rows show the *miss rates* for the 48K and 192K TCs, respectively. The significantly-higher TC miss rates show that the efficiency of the TC rapidly decreases as the number of threads increases.

Table 6 also shows the average number of instructions supplied by a 64K i-cache (*64K IC avg insts*), a 48K TC with its accompanying 64K i-cache (*48K TC avg insts*), and an

Table 6: Trace cache statistics

	ILP workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	4.3	6.9	8.2	9.2
Weighted Speedup (%)	28.5	5.2	-2.3	-12.7
64K IC miss rate (%)	0.1	0.0	0.2	0.2
48K TC miss rate (%)	10.5	20.1	31.2	41.9
192K TC miss rate (%)	1.2	4.2	10.8	19.7
64K IC avg Insts	5.2	8.5	9.4	9.8
48K TC avg insts	8.5	9.2	9.0	8.4
Ideal TC avg insts	8.7	10.8	11.3	10.9
	MEM workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	1.4	2.0	3.1	3.5
Weighted Speedup (%)	4.3	6.1	4.4	0.0
64K IC miss rate (%)	0.0	0.0	0.0	0.0
48K TC miss rate (%)	0.2	0.8	2.5	6.8
192K TC miss rate (%)	0.0	0.1	0.3	0.7
64K IC avg Insts	1.6	2.9	4.8	5.3
48K TC avg insts	2.0	3.5	5.1	5.4
Ideal TC avg insts	2.0	3.5	5.1	5.4
	MIX workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	2.6	4.8	6.9	7.6
Weighted Speedup (%)	15.7	13.6	4.2	-1.2
64K IC miss rate (%)	0.0	0.1	0.1	0.2
48K TC miss rate (%)	5.2	8.6	17.2	29.0
192K TC miss rate (%)	0.6	1.0	4.2	10.0
64K IC avg Insts	3.1	6.2	8.8	9.2
48K TC avg insts	4.4	7.9	9.3	8.9
Ideal TC avg insts	4.4	8.5	10.4	10.6

ideal TC (*Ideal TC avg insts*) to the pipeline. On average, TC can supply 8.5 instructions per cycle in a single thread, which is 63% more than an i-cache can supply. When multiple threads are available, SMT uses the second fetch port to fetch from another thread. Therefore, SMT sustains high instruction throughput without the complication of a TC. With eight threads, the base i-cache with two ports achieves 9.8 IPC, which is higher than TC's. The base case's higher bandwidth combined with the large, diverging gap between the base case's and TC's miss rates as the number of threads increases, clearly shows that TC's space-for-bandwidth trade-off is not effective in SMT.

There is an interesting observation in Figure 5: MIX gets more benefit from TC than ILP and MEM as threads increase. As expected, ILP gets the most benefit of TC in single-thread runs. As threads increase, the pressure on the TC greatly increases and the miss rate in the TC increases quickly. When I put ILP and MEM together (MIX), the ILP threads experience less pressure in the TC compared to the ILP threads in the ILP workload because ILP threads in MIX take up the slack of the TC space created by MEM threads in MIX. This argument is supported by TC's miss rate shown in Table 6. For instance, TC's miss rate for eight threads in MIX is similar or lower than TC's miss rate for four threads in ILP.

Because my experiments show negative results for TC, I ensure that the negative results are not caused by the utilization for the SMT base case that is overly high such that it leaves no headroom for TC to improve. To that end, , I measure the utilizations for the 16-issue SMT base case. The utilization is defined as the number of instructions fetched, regardless of whether it is committed or squashed later in the pipeline, divided by the execution time. The utilizations for the 16-issue SMT base case used in the TC experiments with eight threads are 9.7, 3.9 and 8.1 for ILP, MEM and MIX workloads, recpectively.

Some processors use TC to hold pre-decoded instructions (e.g. Pentium IV). If such a cache holds merely decoded individual instructions but not traces spanning multiple branches, I consider such a cache to be an i-cache and not a TC, and my results are not applicable to it. In an SMT that already includes TC, the OS and hardware can turn off TC by simply aligning instructions in the TC as they would be in an IC and allow accesses to

start in the middle of cache blocks. The latter is difficult for TC because the PC of an instructions inside a cache line, other than the first instruction, is unknown and cannot be easily obtained.

My experiments favor TC by giving it unrealistic advantages and an aggressive, 16-issue processor which gives TC much headroom for improvement. Nevertheless, I find that TC degrades SMT throughput. Using miss latencies longer than my numbers to model future technology will shift the performance bottleneck to the back-end and reduce opportunity for the front-end, further discouraging the use of TC. I also show that an ideal TC only marginally improves throughput. Therefore, my results unequivocally prove that TC hurts SMT running more than two threads, and there is no need to vary other parameters.

Figure 6 through Figure 21 shows PF's speedup and IPC for each workload, as well as the speedup and IPC component in the multi-programmed workloads. These graphs were summarized and shown in Figure 5. In each figure, the top graph shows the speedup of each thread in a workload and the bottom graph shows the IPC of each thread in a workload. The sum of IPCs of these thread in a workload forms the total throughput of the workload.

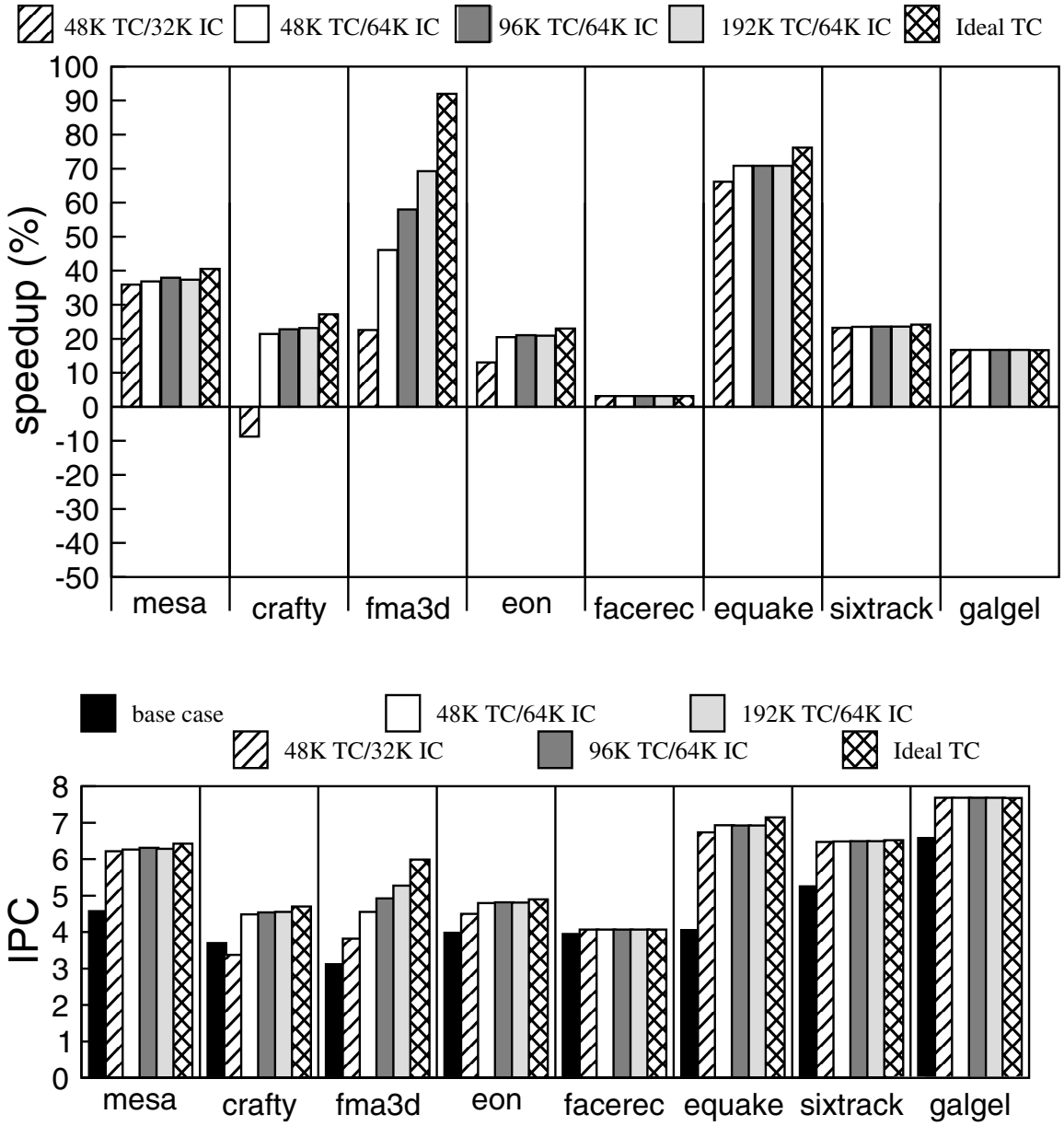


Figure 6: Detailed IPC and speedup of TC for 1T.ILP

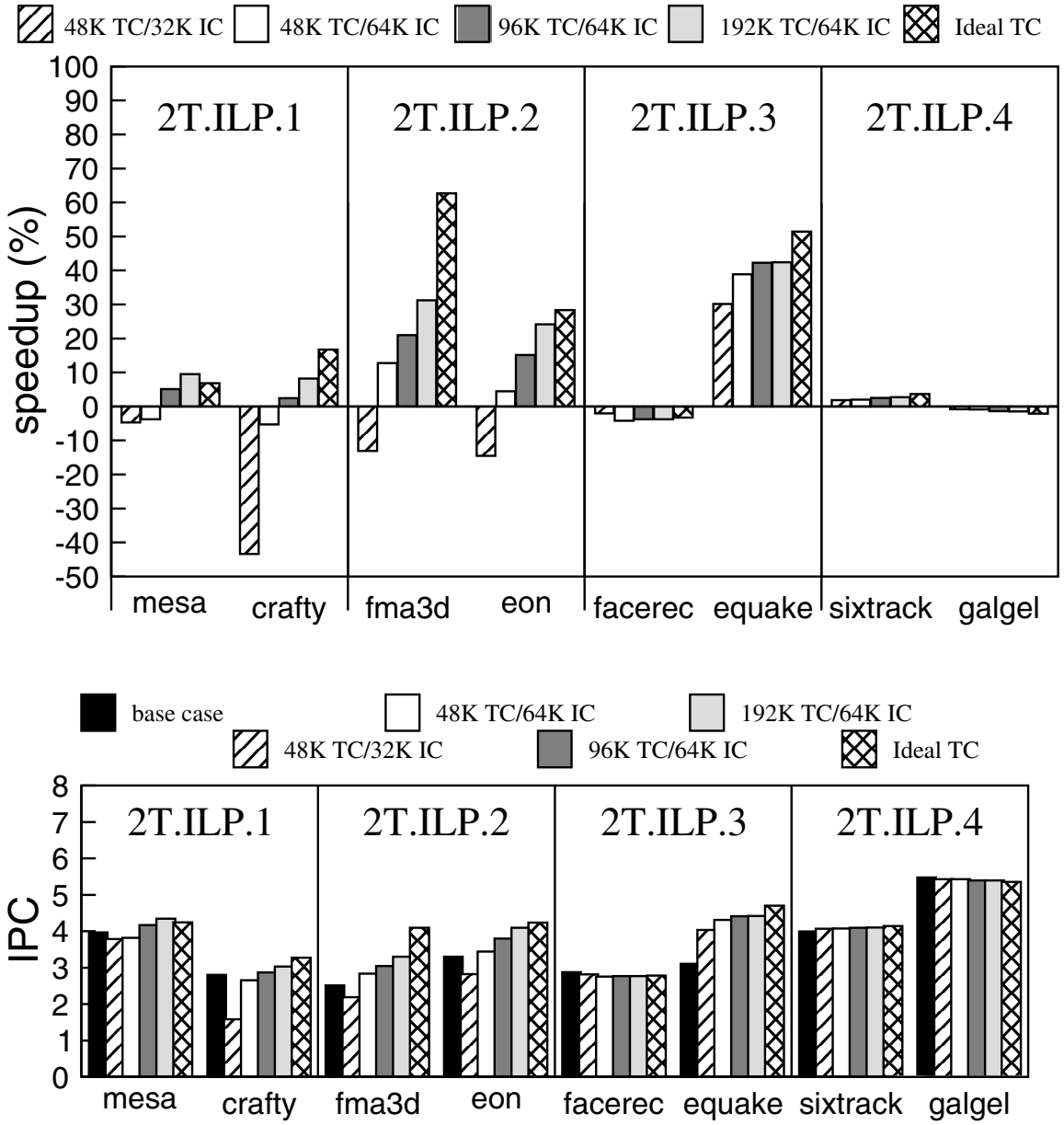


Figure 7: Detailed IPC and speedup of TC for 2T.ILP

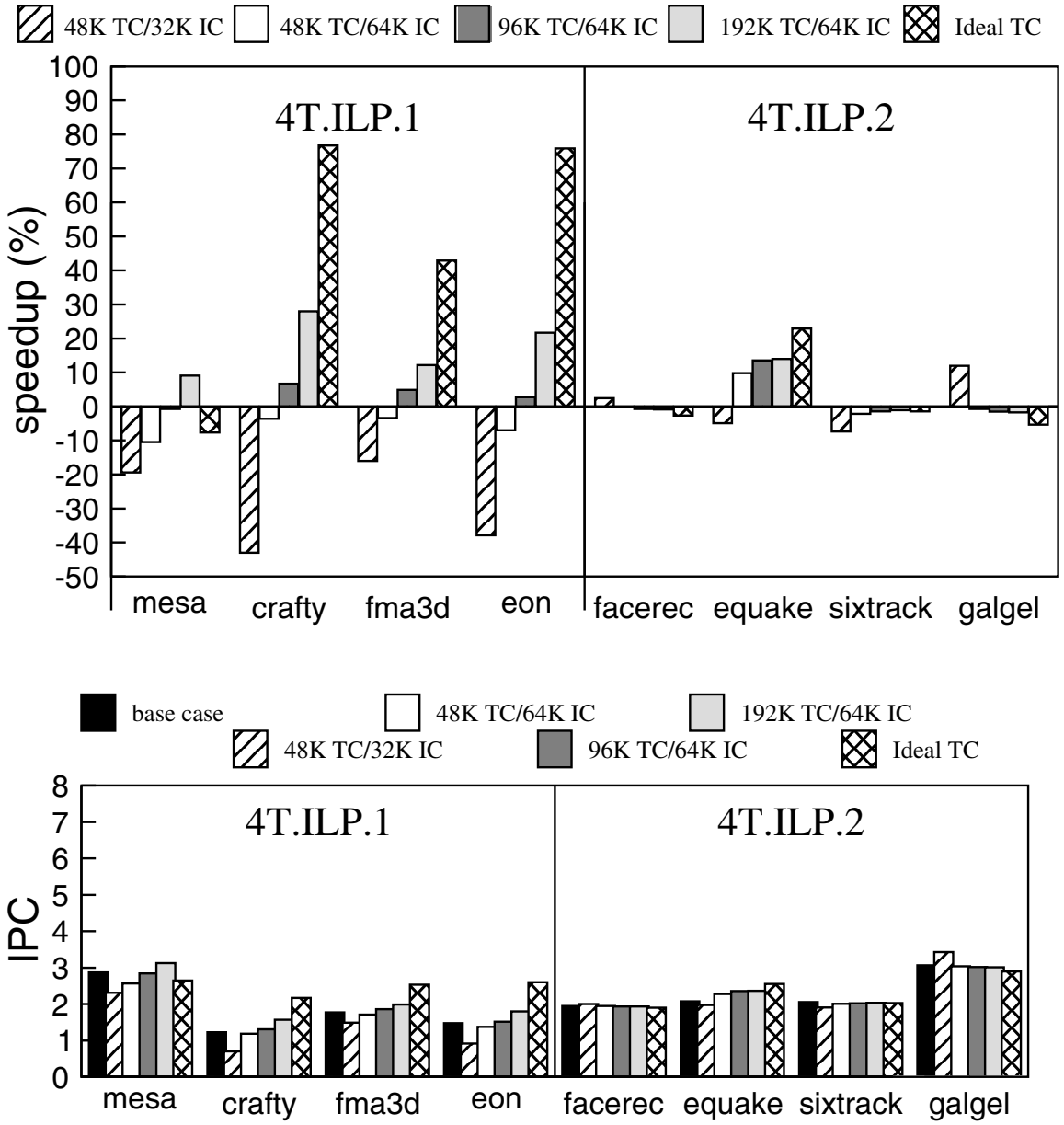


Figure 8: Detailed IPC and speedup of TC for 4T.ILP

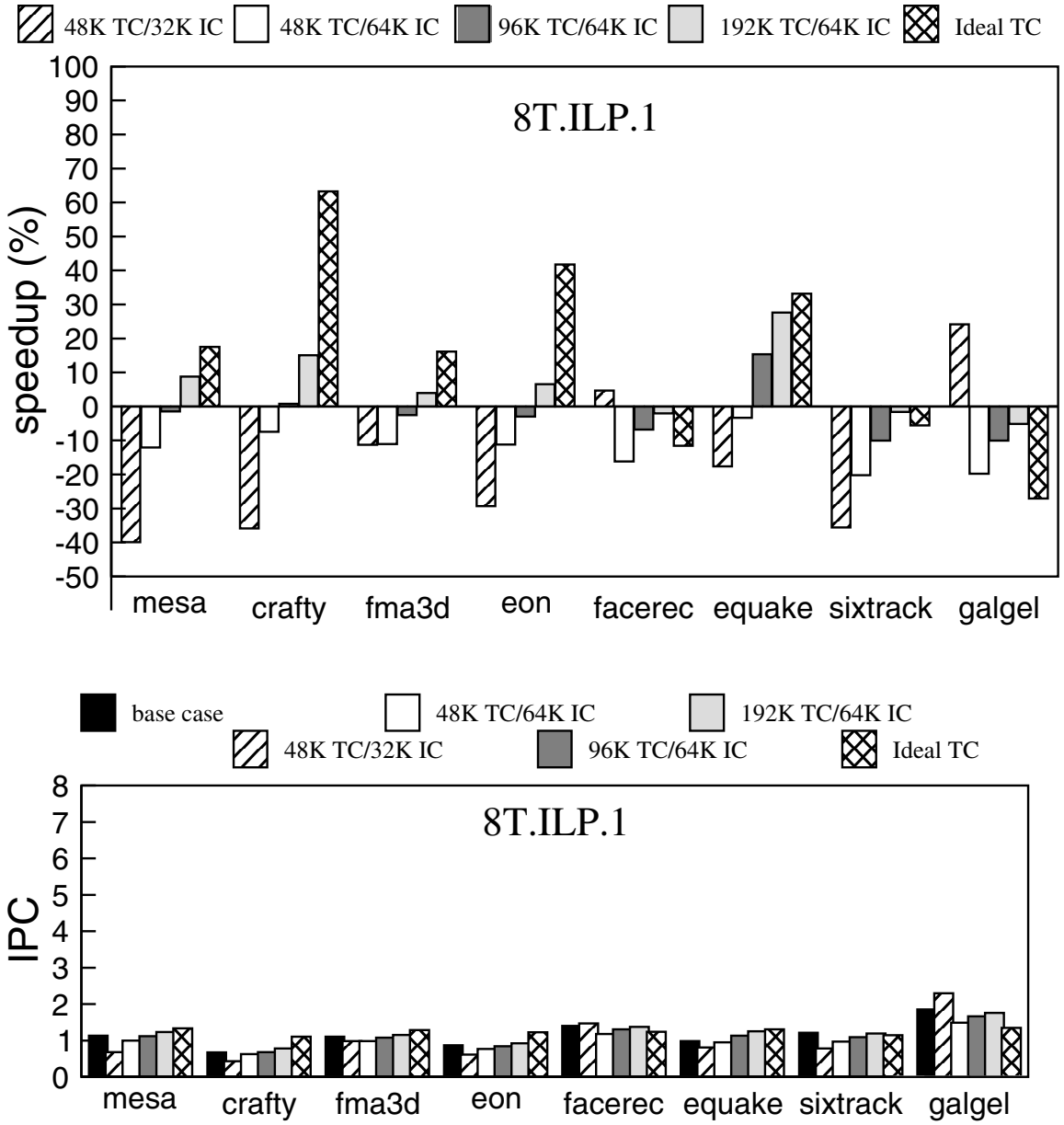


Figure 9: Detailed IPC and speedup of TC for 8T.ILP

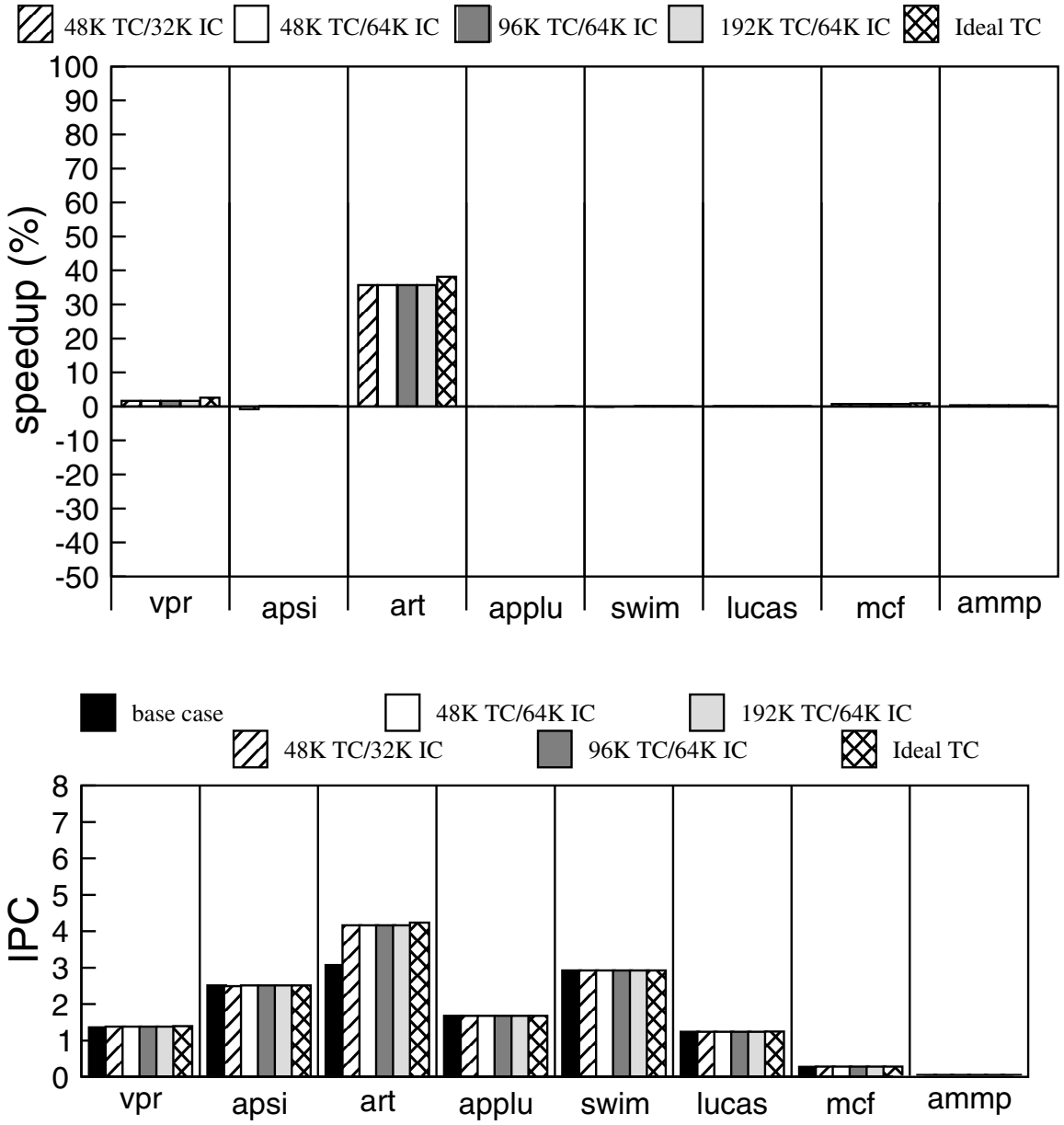


Figure 10: Detailed IPC and speedup of TC for 1T.MEM

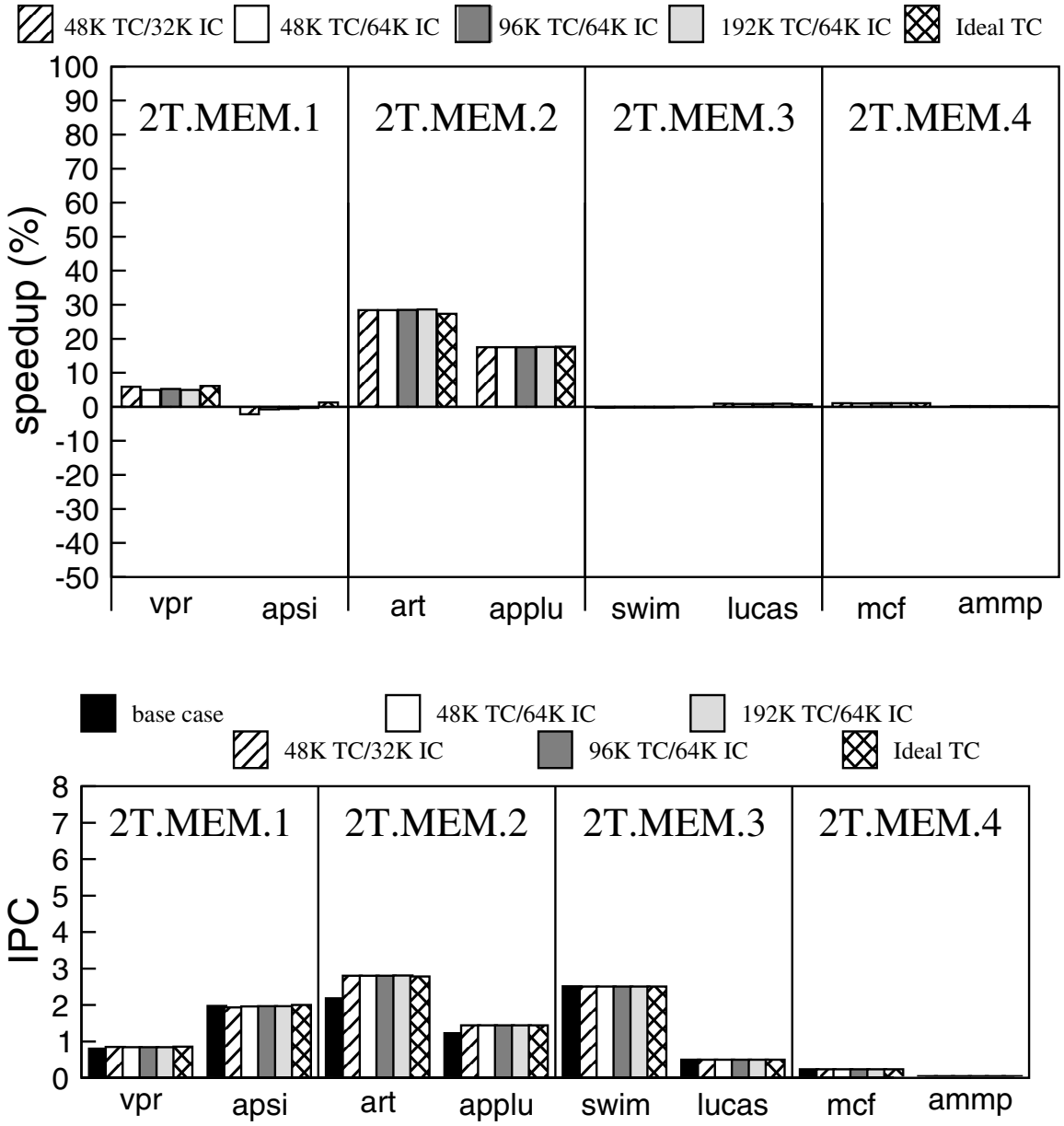


Figure 11: Detailed IPC and speedup of TC for 2T.MEM

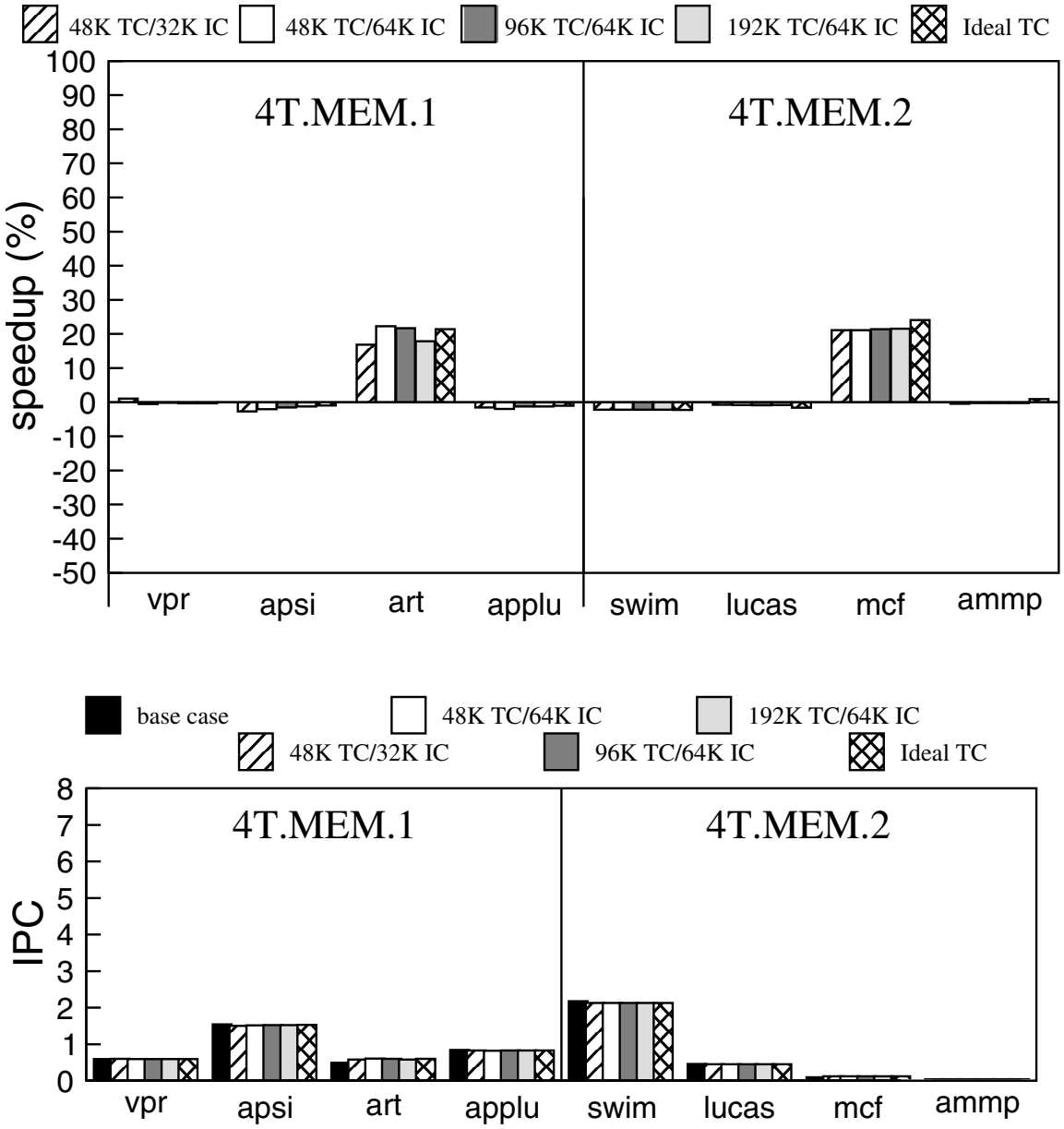


Figure 12: Detailed IPC and speedup of TC for 4T.MEM

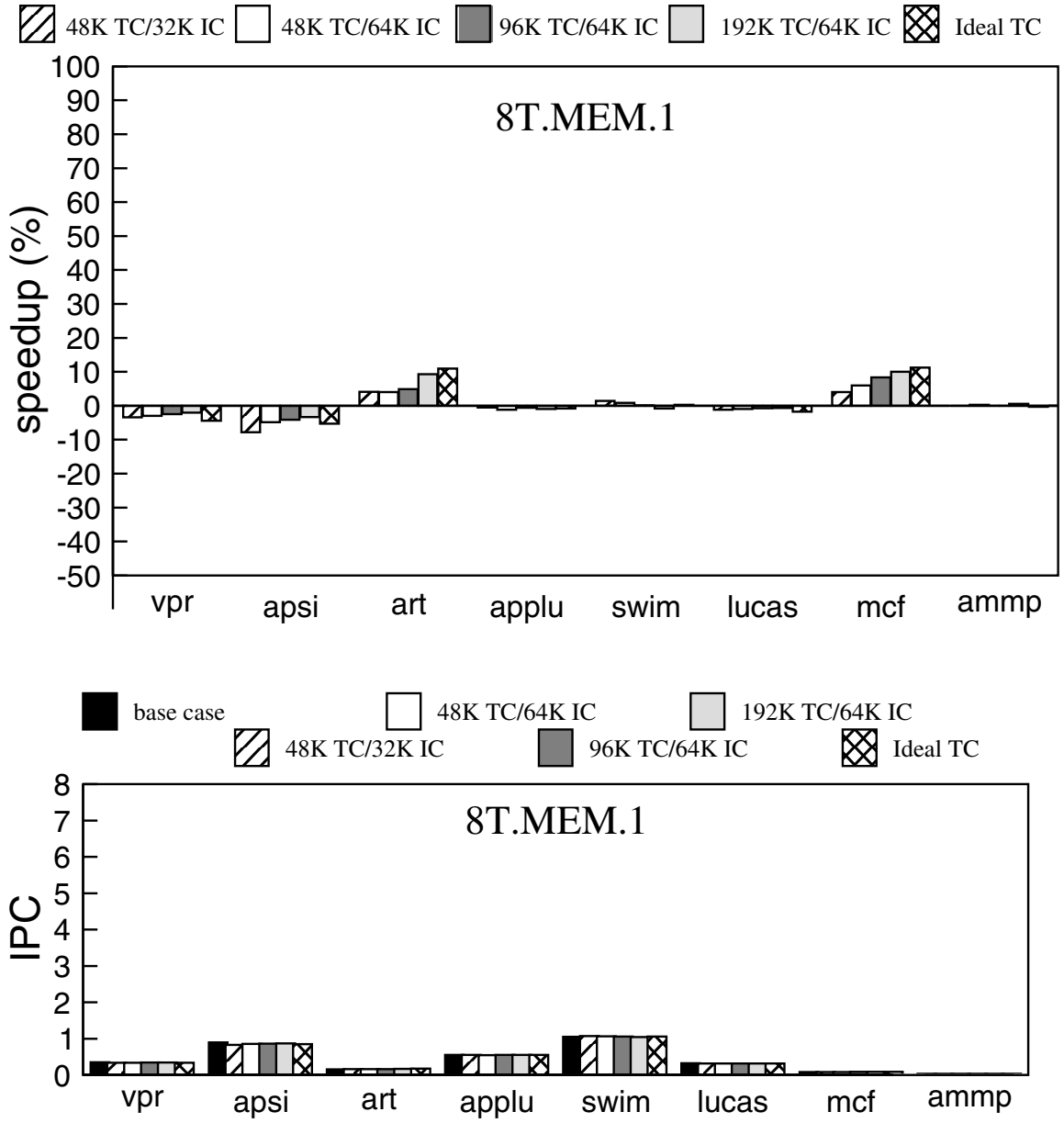


Figure 13: Detailed IPC and speedup of TC for 8T.MEM

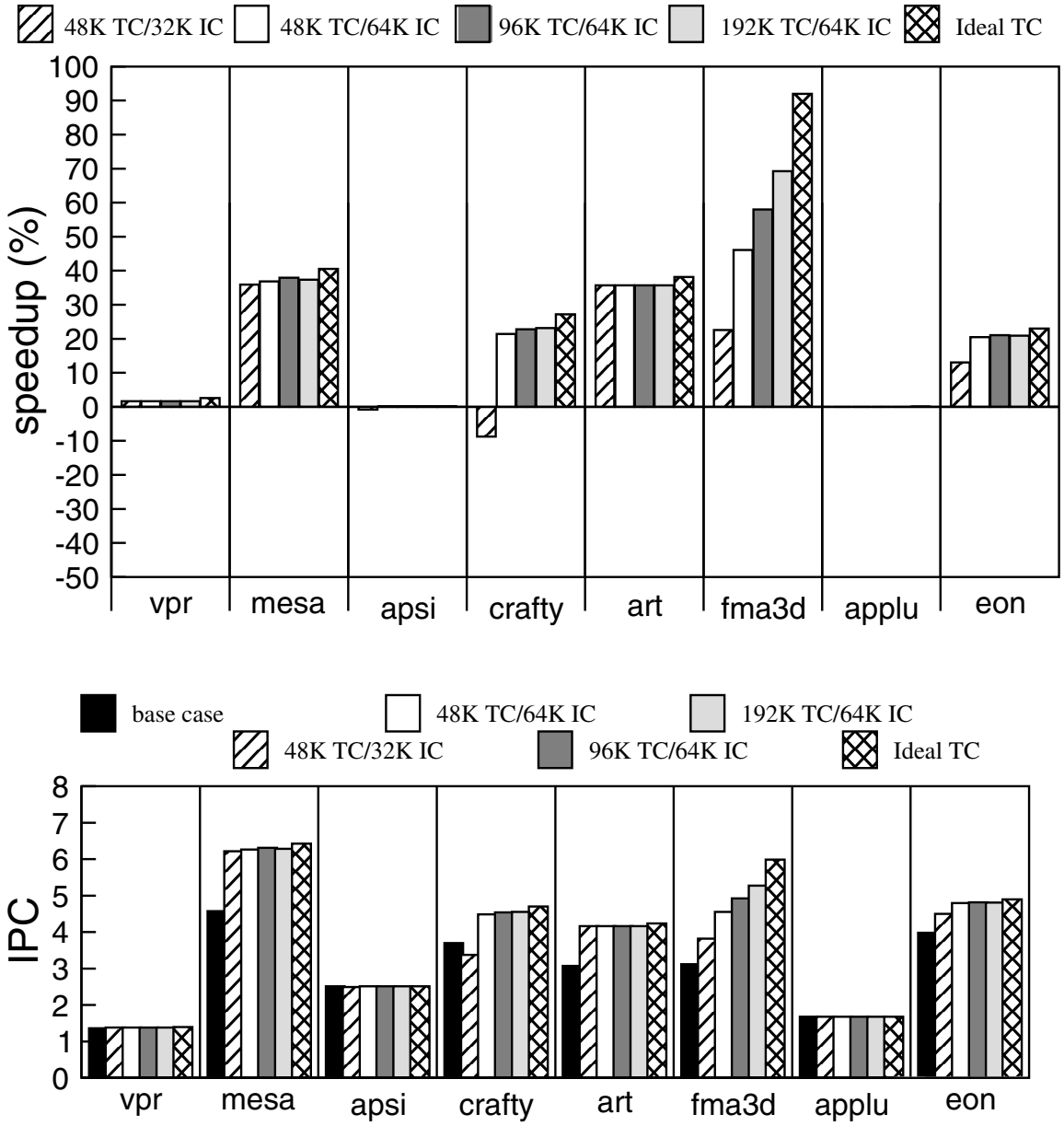


Figure 14: Detailed IPC and speedup of TC for 1T.MIX (part I)

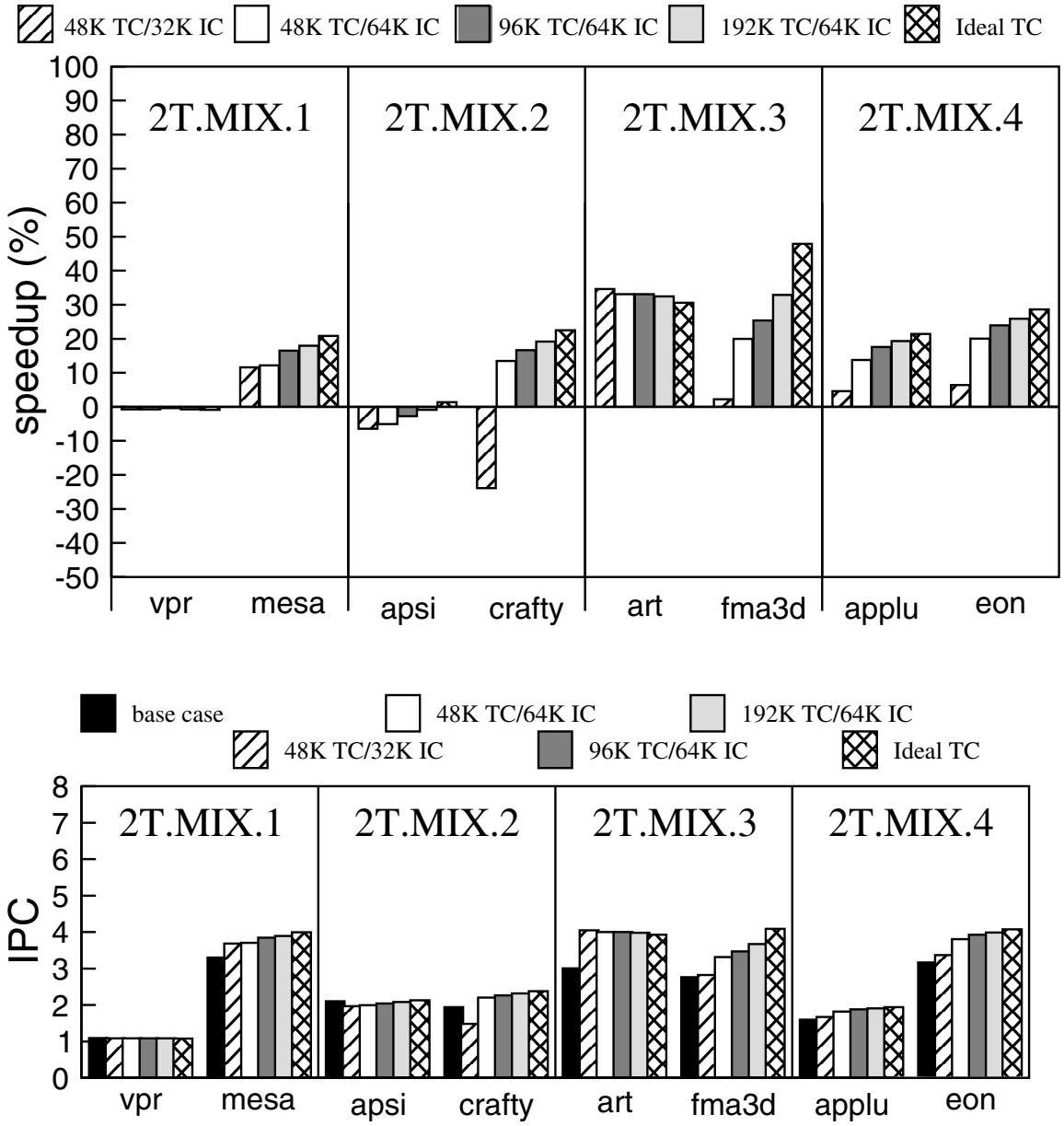


Figure 15: Detailed IPC and speedup of TC for 2T.MIX.{1,2,3,4}

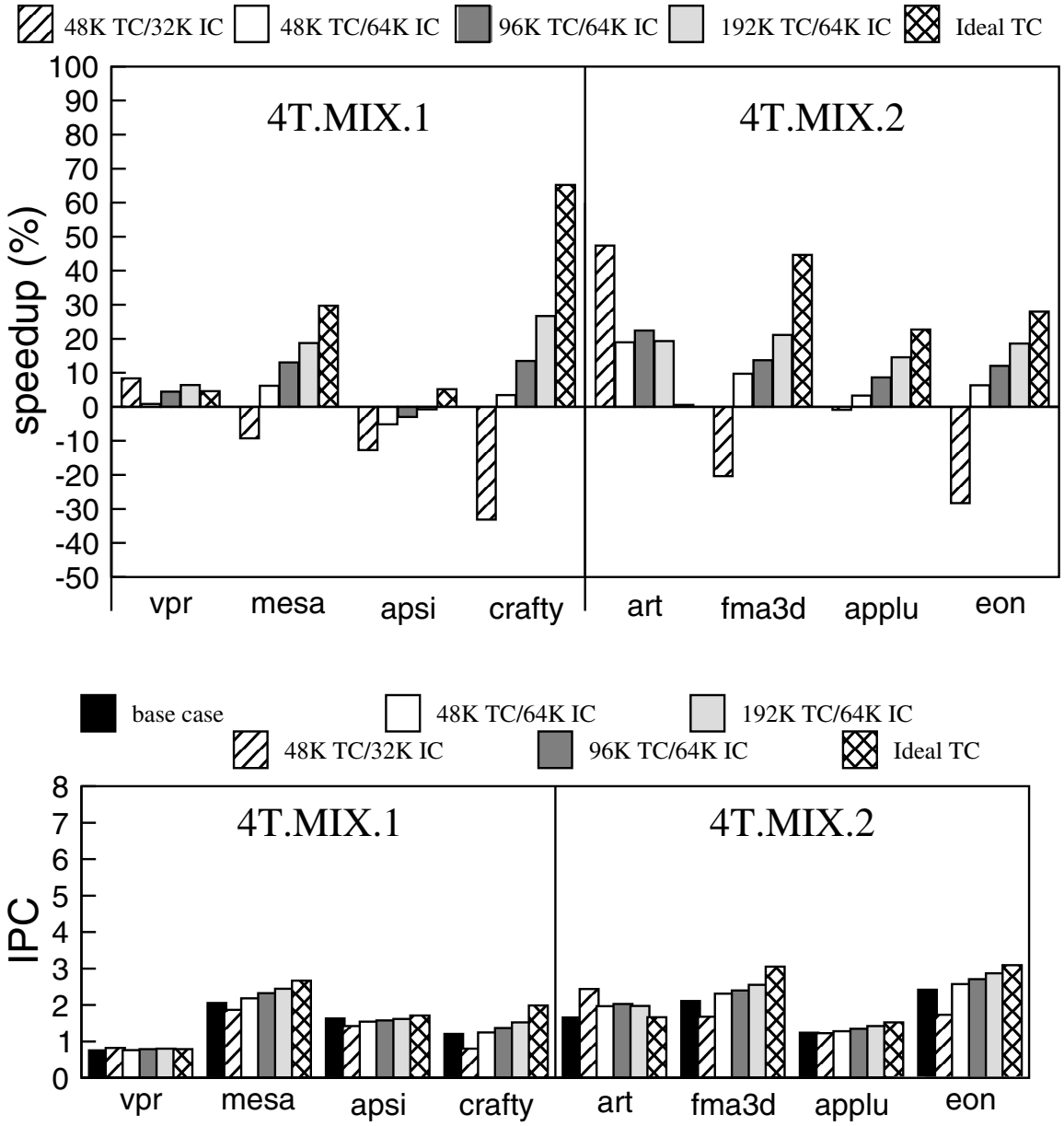


Figure 16: Detailed IPC and speedup of TC for 4T.MIX.{1,2}

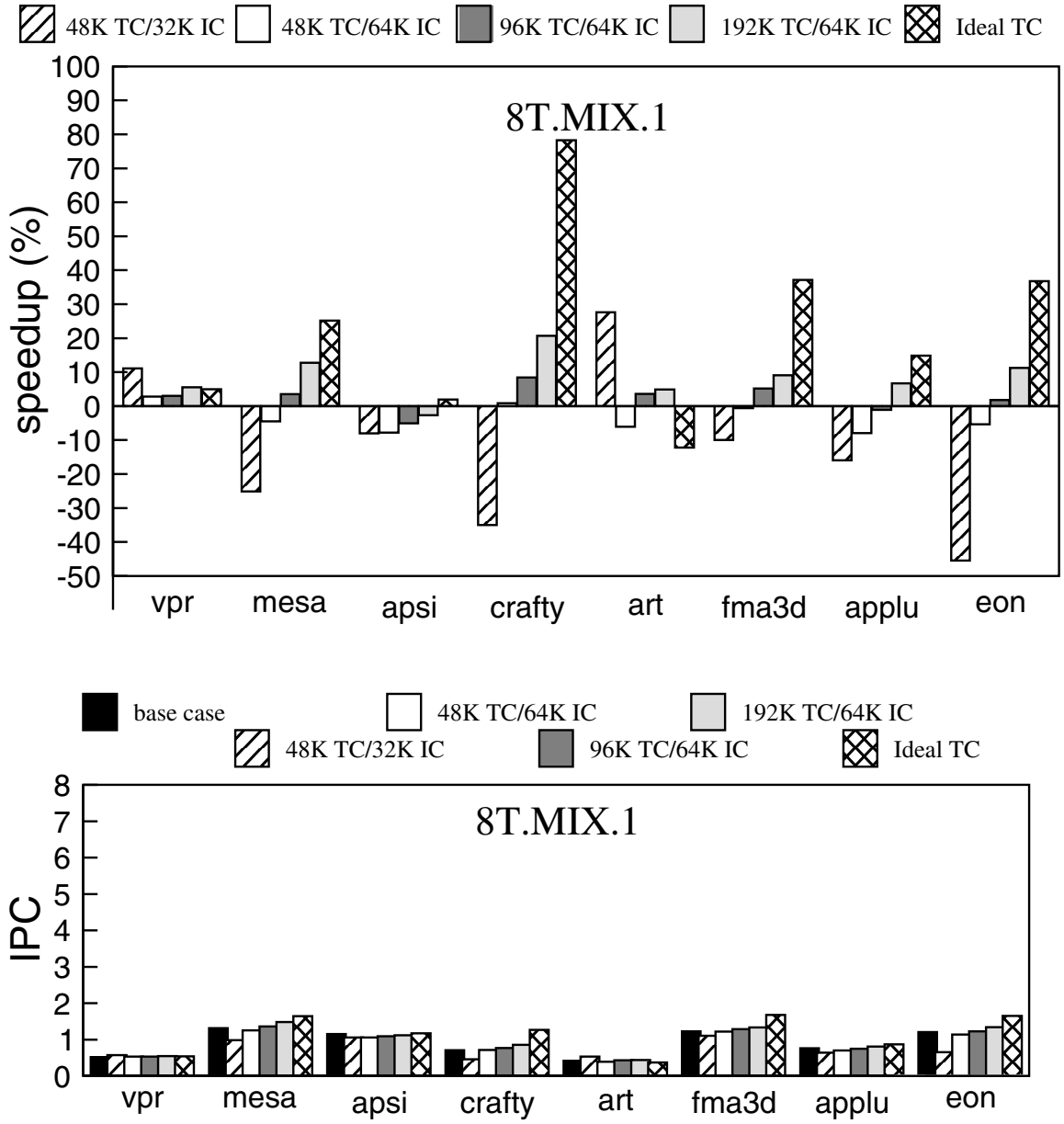


Figure 17: Detailed IPC and speedup of TC for 8T.MIX.1

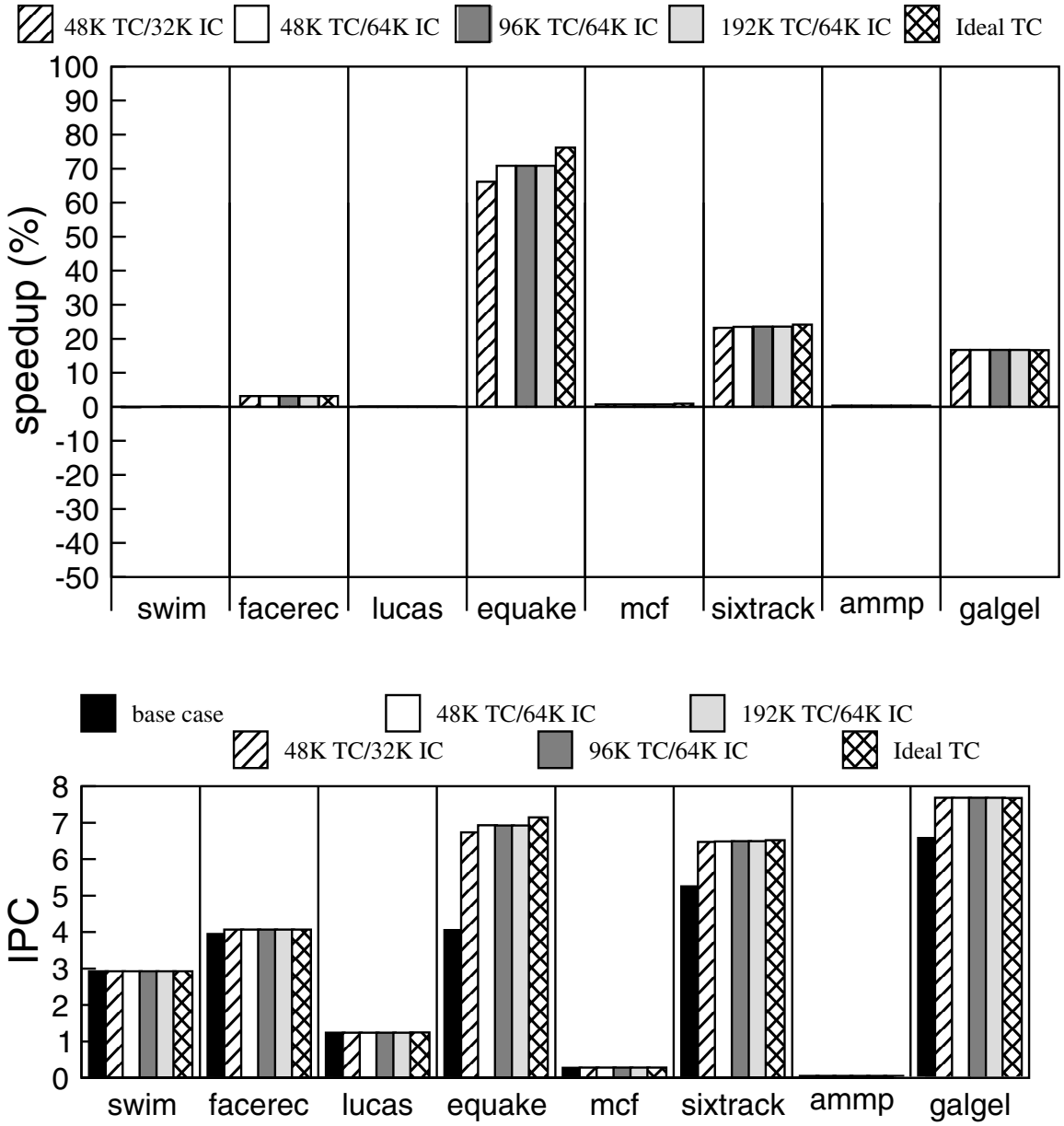


Figure 18: Detailed IPC and speedup of TC for 1T.MIX (part II)

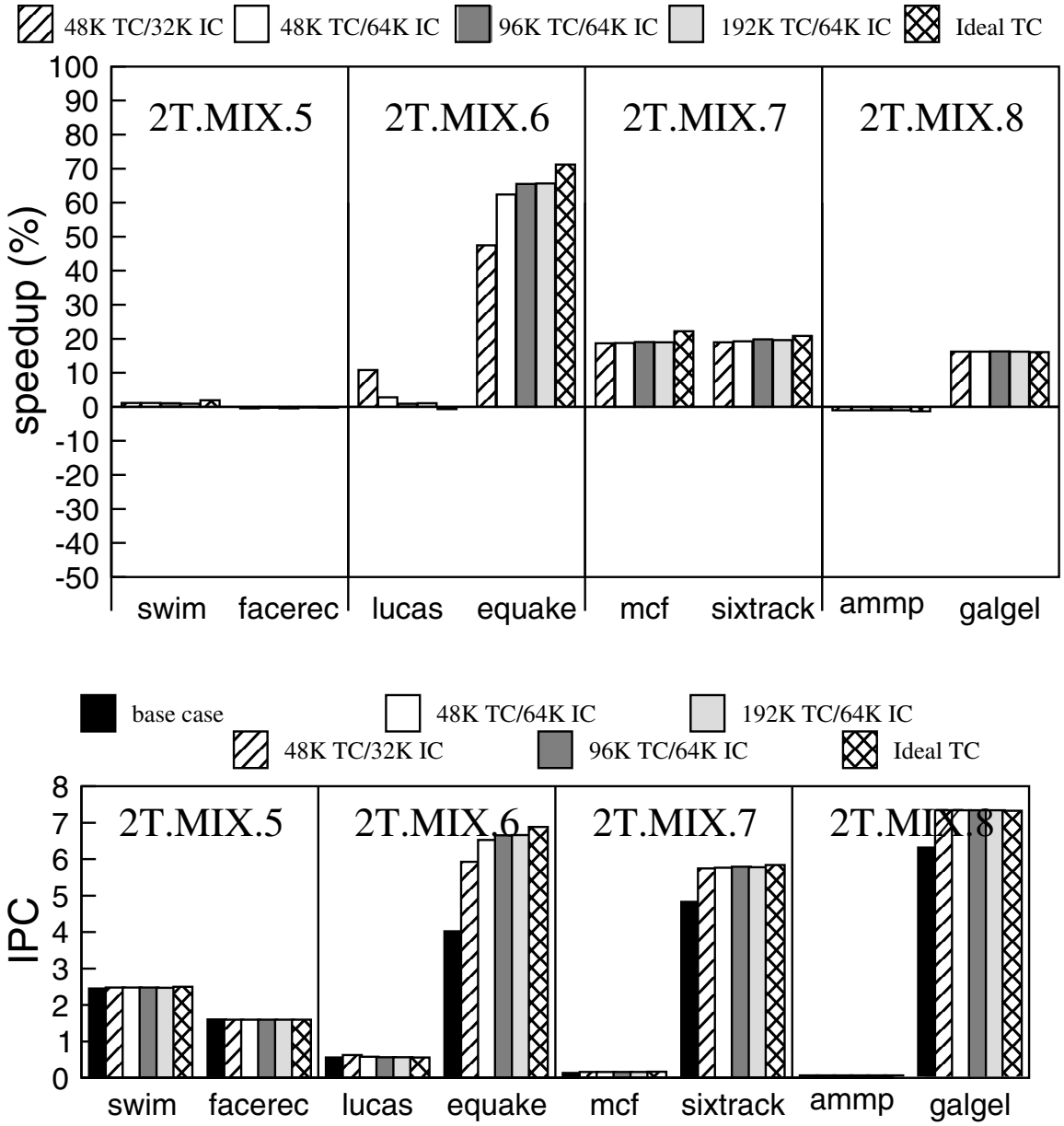


Figure 19: Detailed IPC and speedup of TC for 2T.MIX.{5,6,7,8}

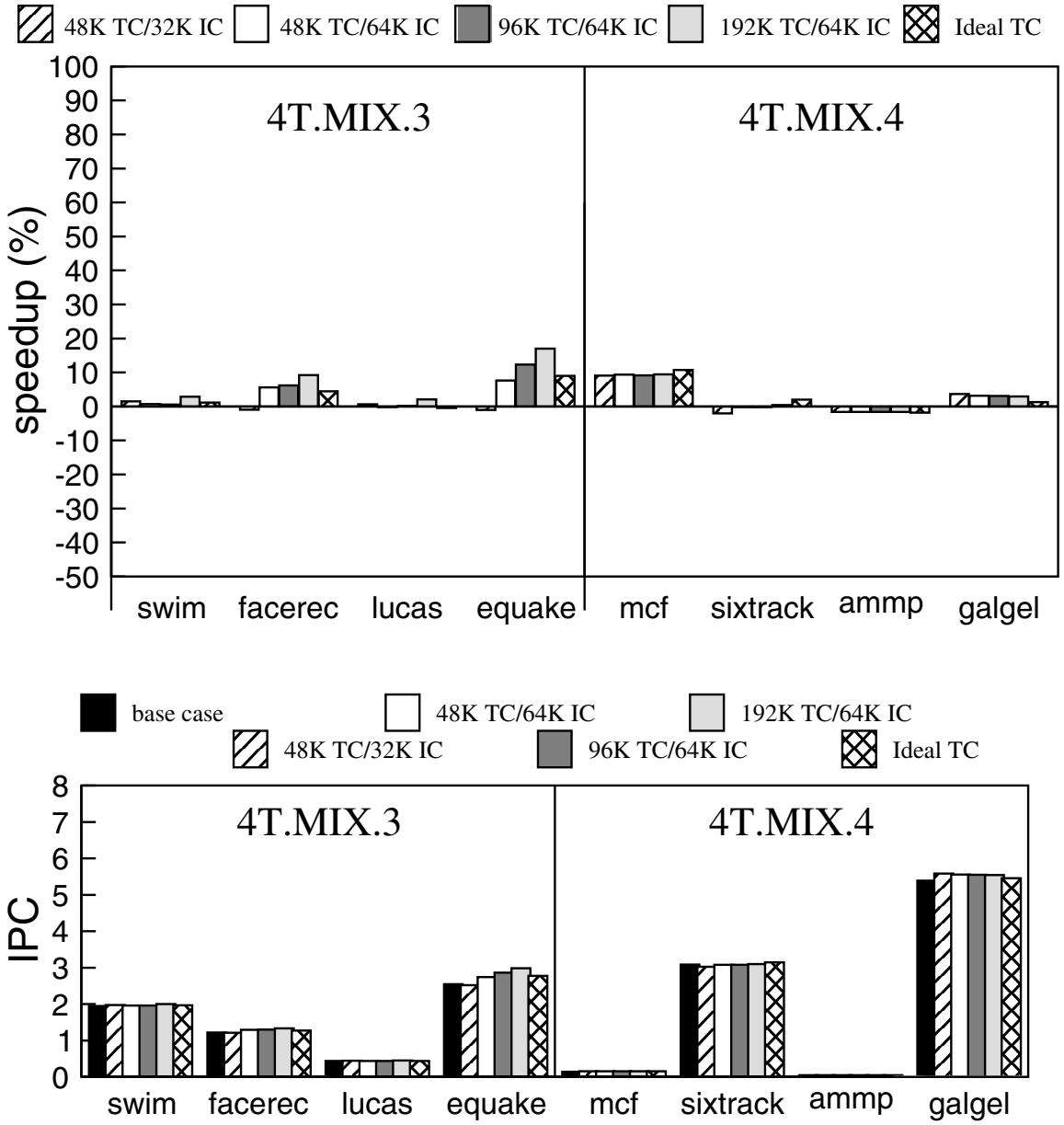


Figure 20: Detailed IPC and speedup of TC for 4T.MIX.{3,4}

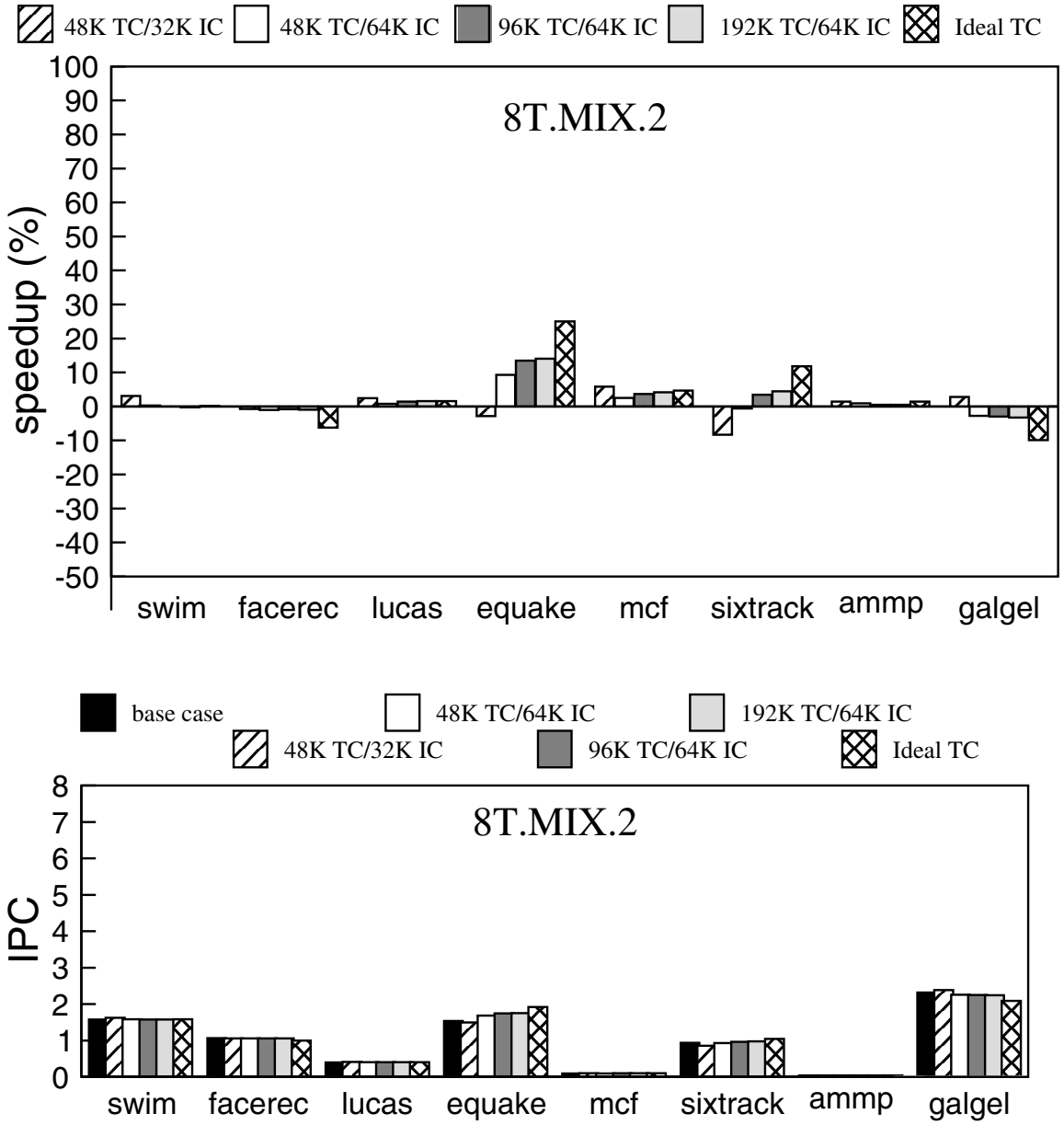


Figure 21: Detailed IPC and speedup of TC for 8T.MIX.2

5.2 Value Prediction Results

Recall from Section 1 that value-predicting a long-latency operation causes hold-up of registers until the operation completes and the prediction is confirmed. This hold-up occurs even with correct value prediction. In contrast, SMT simply squashes the thread containing the operation, releasing the resources held by the thread and allowing other threads to progress. Thus, there is a choice of value-predicting and holding up registers, versus squashing and overlapping the latency with other threads. SMT's sharing of registers among its threads impacts this choice. In this section, I evaluate this choice in SMT.

Each thread has two four-way, 8K-entry tables, one each for stride prediction and context prediction. To minimize mispredictions, each of these table also has its own 2KB confidence tables. The total size of the VP tables in an eight-thread SMT is a generous 5MB, ensuring that my results are not limited by small tables.

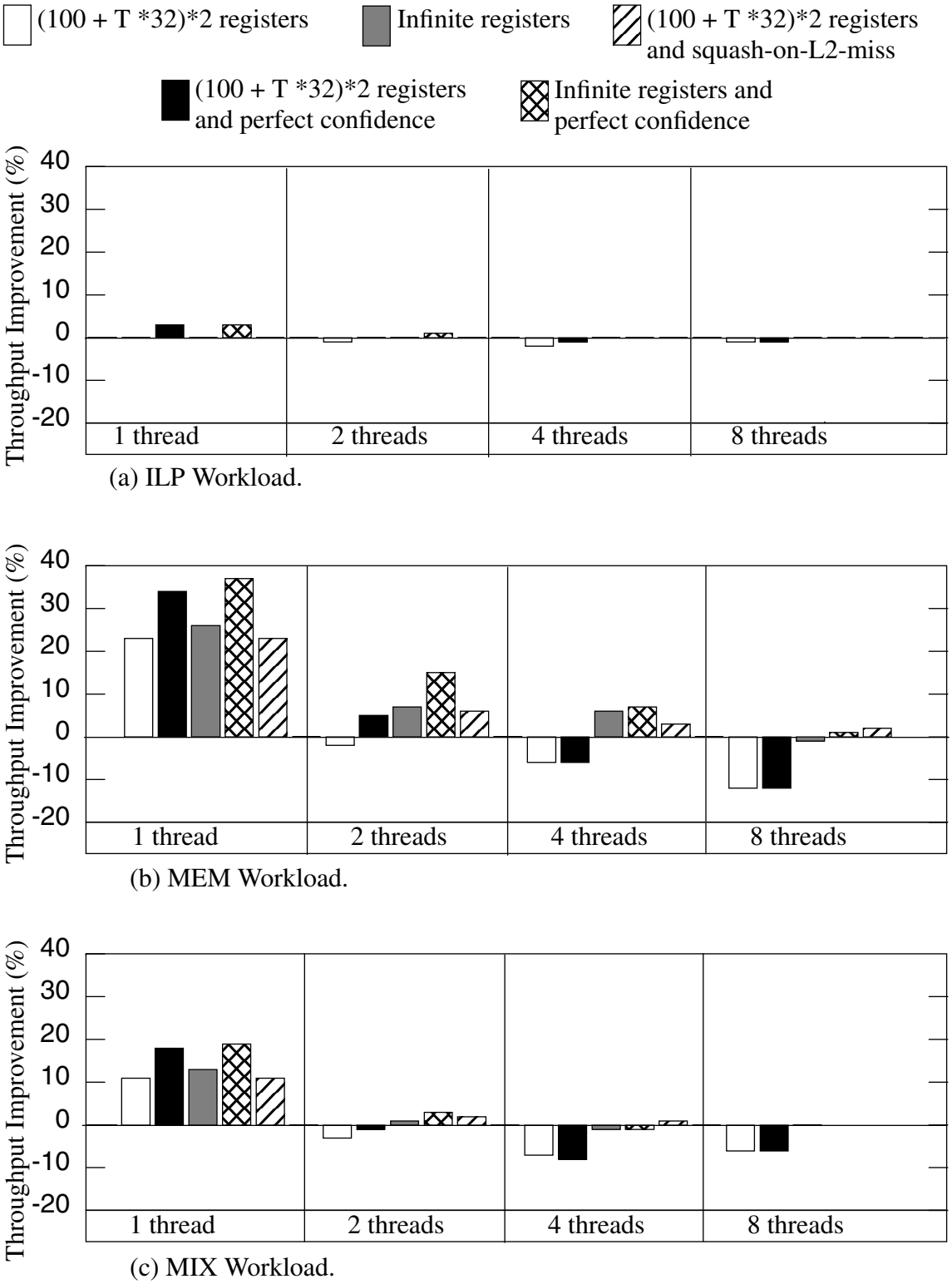
Figure 22 shows VP's throughput improvements compared to an SMT without VP. Similar to Section 5.1, Figure 22 shows three sets of workloads, ILP, MEM, and MIX, and varies the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set.

Because I am interested in register pressure in the presence of VP, I show two configurations with different number of physical registers. One configuration, called *VP-finite*, contains (100 + T * 32) integer registers and (100 + T * 32) floating-point registers, where T is the number of threads. The number 32 in this expression is the minimum number required for per-thread architectural registers, and 100 is the number of registers for renaming. This configuration represents a realistic number of registers, and it has also been used in [48]. The other configuration, called *VP-infinite*, has infinite physical registers. To examine if VP can benefit SMT by overlapping only L1 misses, I show a configuration called *VP-squash*. *VP-squash* uses VP if a load misses in L1 but squashes (mentioned in Section 4) if the load also misses in L2. *VP-squash* has the same number of physical registers as *VP-finite*.

Figure 22 shows a group of five bars for a given number of threads. The first bar shows *VP-finite*. The second bar shows *VP-finite* with perfect confidence prediction. The third and fourth bars show *VP-infinite* without and with perfect confidence prediction, respectively, quantifying VP's potential if register pressure were absent. The base case for the first and second bars is an SMT with as many physical registers as *VP-finite*. The base case for the third and fourth bars is an SMT with an infinite number of physical registers. The last bar shows *VP-squash* normalized to the base case for *VP-finite*.

Figure 22 shows that VP benefits MEM and MIX but not ILP. This result is hardly surprising because VP is triggered only for L1 misses, and ILP has low L1 miss rates. On the other hand, VP hides the penalties of the misses present in MEM and MIX. Looking at MEM and MIX, this figure shows a interesting trend: VP significantly improves single-thread performance, especially for MEM. This result agrees with the results from previous VP papers [30,2,40,32,9], indicating that my value predictor is implemented correctly. However, *VP-finite*'s throughput improvements decrease significantly and become negative as the number of threads increases in both MEM and MIX. Note that the *base case throughput*, shown in the first row of Table 7, continues to improve as I increase the number of threads to eight, showing that VP's diminishing returns are *not* due to pipeline saturation. *VP-finite* with perfect confidence (second bar) shows the same trend, showing that the degradation exists even when VP is 100% accurate (albeit at non-perfect coverage). Thus, the second bars rule out mispredictions as the cause of the degradation trend.

Two reasons contribute to VP's degradation with multiple threads, even with large VP tables. First, VP's holding up of registers degrades throughput with two or more threads. Figure 22 support this observation by showing that *VP-finite*'s degradation largely disappears when infinite registers are available, as shown by *VP-infinite* (third and fourth bars). Second, SMT's latency tolerance reduces VP's opportunity. Figure 22 supports this argument by showing that even using *VP-infinite* with perfect confidence, VP's opportunity diminishes and eventually disappears, as the number of threads increases. Near-zero opportunity combined with register pressure forces *VP-finite* to incur degradation at more



(a) ILP Workload.

(b) MEM Workload.

(c) MIX Workload.

Figure 22: Value Prediction throughput improvements.

than two threads. The near-zero opportunity also shows that VP's benefit would be marginal, even if the VP implements selective recovery (mentioned in Section 2.2) to reduce misprediction penalty.

The four-thread *VP-infinite* bars for MIX show a small degradation despite having infinite registers. Because MIX has high-ILP and low-ILP threads, VP's impact is not uniform (this point is also made in Section 5.1). VP helps low-ILP threads more than high-ILP threads to the point that SMT allocates resources to the improved-but-still-low-ILP threads at the cost of the high-ILP threads. Such allocation causes a slight overall throughput degradation. *VP-squash* improves two-thread MEM by 6%, but improves little for other workloads (0-3%). This result shows that implementing VP to overlap only L1 misses is not profitable for SMT

Table 7 shows important statistics for VP. *Coverage* is the ratio of the number of predictions over the number of L1 load misses. *Squash Rate* is the ratio of the number of squashes caused by value mispredictions over the number of predictions. We see that coverage and squash rate are fairly stable across threads, and the squash rate is low. The stability of these metrics clearly indicates that VP's degradation with two or more threads is not due to worse coverage or more value mispredictions.

The fourth, fifth and sixth rows show the *weighted speedup* (explained in Section 5.1) for *VP-finite*, *VP-infinite and VP-squash*. *VP-finite*'s weighted speedups are positive for two or more MEM and MIX threads while the overall throughput degrades. Because there is a large variance in the individual predictability and IPCs of these MEM and MIX threads, VP's impact is uneven among the threads, causing weighted speedup to deviate from overall throughput (as explained in Section 5.1). VP fools SMT into allocating more registers to the improved-but-still-low-ILP threads which hold up the registers from the high-ILP threads, degrading overall throughput. Because SMT's goal is to improve processor throughput, techniques which improve individual threads while degrading processor throughput defeat SMT's purpose. In fact, SMT does the reverse: SMT employs several optimizations which improve processor throughput at the cost of individual threads. For example, (1) because the SMT pipeline is typically deeper than a superscalar

pipeline [49], single-thread performance slightly worsens on SMT. (2) SMT's ICOUNT, which optimizes processor throughput, may worsen a low-IPC thread by fetching more often from higher-IPC threads [48]. (3) Squashing a thread on L2 misses improves processor throughput while slightly worsening the thread's IPC [47].

Similar to previous section, because my experiments show negative results for VP, I ensure that the negative results are not caused by the utilization for the SMT base case that is overly high such that it leaves no headroom for VP to improve. To that end, , I measure the utilizations for the eight-issue SMT base case. The utilization is defined as the number of instructions fetched, regardless of whether it is committed or squashed later in the pipeline, divided by the execution time. The utilizations for the eight-issue SMT base case used in the VP experiments with eight threads are 5.6, 4.2 and 5.5 for ILP, MEM and MIX workloads, recpectively.

Table 7: Value prediction statistics

	ILP workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	3.5	4.6	5.1	5.3
Coverage(%)	15.0	22.6	27.9	42.2
Squash Rate(%)	0.3	0.2	0.2	0.2
VP-finite Weighted Speedup (%)	0.5	-1.5	-1.6	-0.9
VP-infinite Weighted Speedup (%)	0.5	-0.4	-0.2	-0.4
VP-squash Weighted Speedup (%)	0.5	-0.4	-0.3	-0.4
	MEM workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	1.1	1.8	2.9	3.9
Coverage(%)	34.4	28.4	24.0	31.0
Squash Rate(%)	0.6	0.5	0.4	0.4
VP-finite Weighted Speedup (%)	23	8.8	8.5	7.5
VP-infinite Weighted Speedup (%)	26.0	18.2	19.3	14.7
VP-squash Weighted Speedup (%)	23	12.2	4.4	2.9
	MIX workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	2.1	3.8	4.7	5.2
Coverage(%)	24.3	30.6	28.7	31.3
Squash Rate(%)	0.4	0.3	0.2	0.2
VP-finite Weighted Speedup (%)	11.2	6.0	0.7	1.6
VP-infinite Weighted Speedup (%)	12.5	10.6	7.1	6.9
VP-squash Weighted Speedup (%)	11.3	4.5	2.3	0.9

My experiments favor VP by giving it unrealistic advantages and an aggressive processor which gives VP much headroom for improvement. Still, VP degrades SMT throughput. Because VP holds up physical registers during an L2 miss, using longer miss latencies to model future technology will further degrade throughput. I also show that VP does not improve throughput even with infinite registers. Therefore, my results unequivocally prove that VP hurts SMT and there is no need to vary other parameters.

Figure 23 through Figure 38 shows PF's speedup and IPC for each workload, as well as the speedup and IPC component in the multi-programmed workloads. These graphs were summarized and shown in Figure 22. In each figure, the top graph shows the speedup of each thread in a workload and the bottom graph shows the IPC of each thread in a workload. The sum of IPCs of these thread in a workload forms the total throughput of the workload.

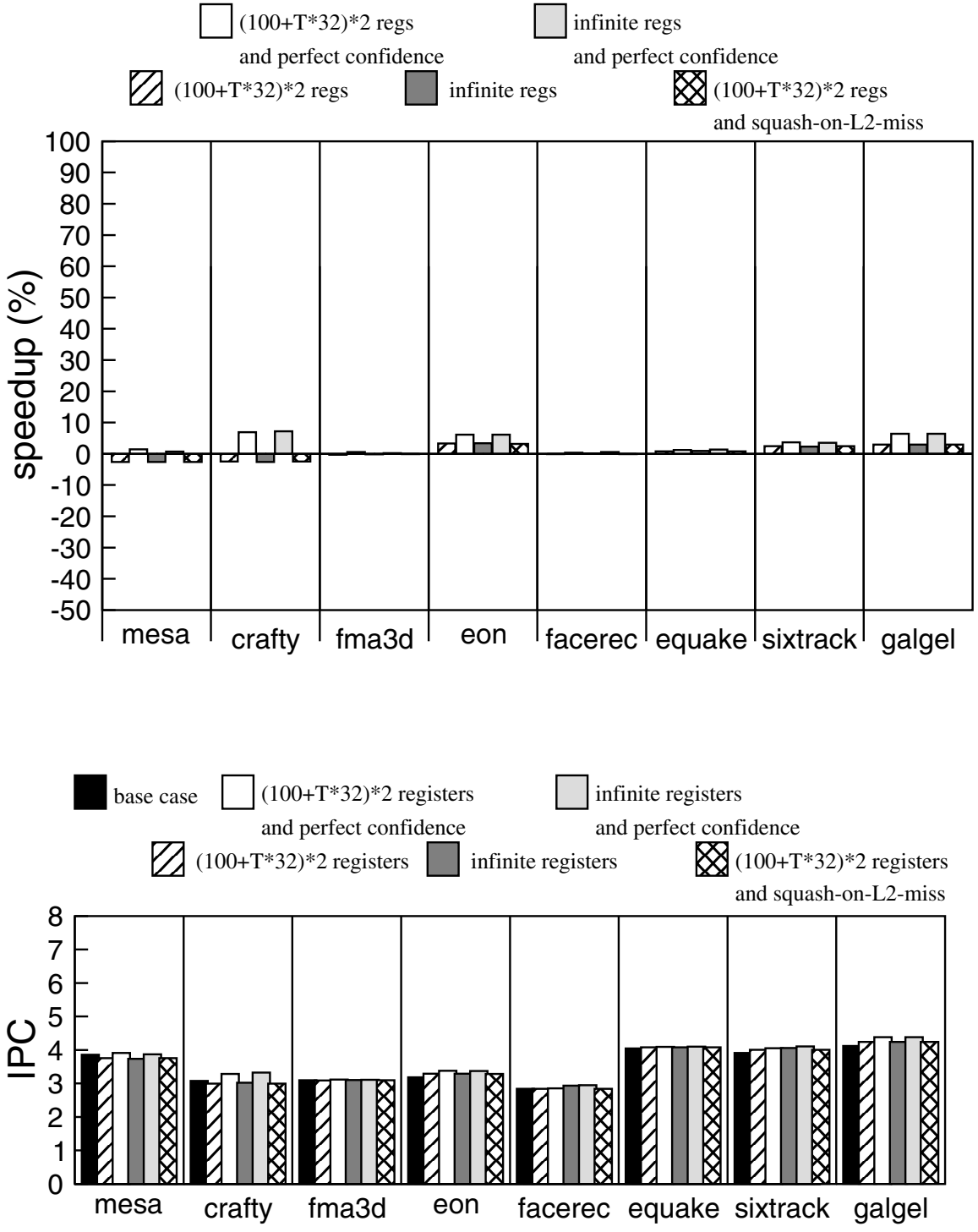


Figure 23: Detailed IPC and speedup of VP for 1T.ILP

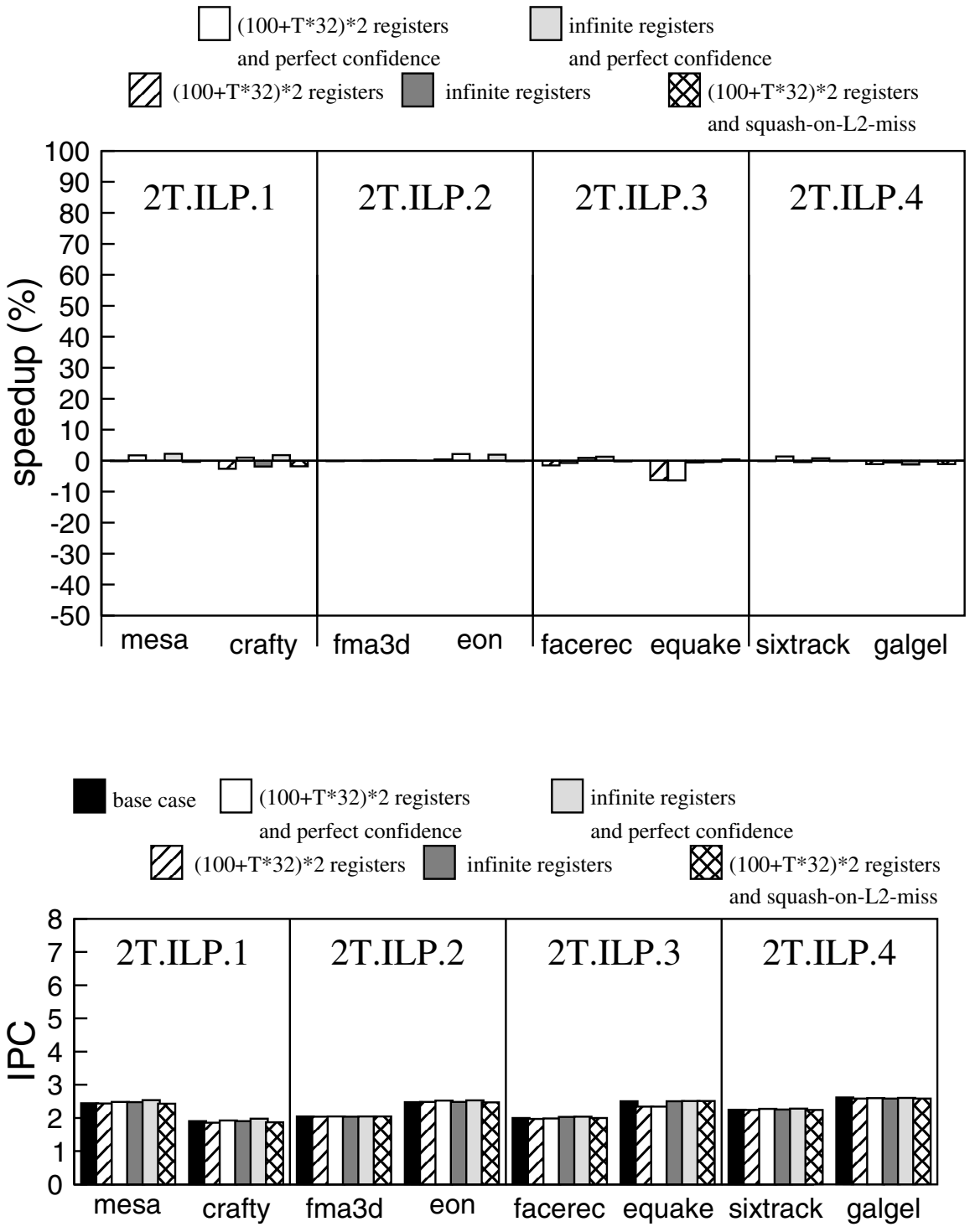


Figure 24: Detailed IPC and speedup of VP for 2T.ILP

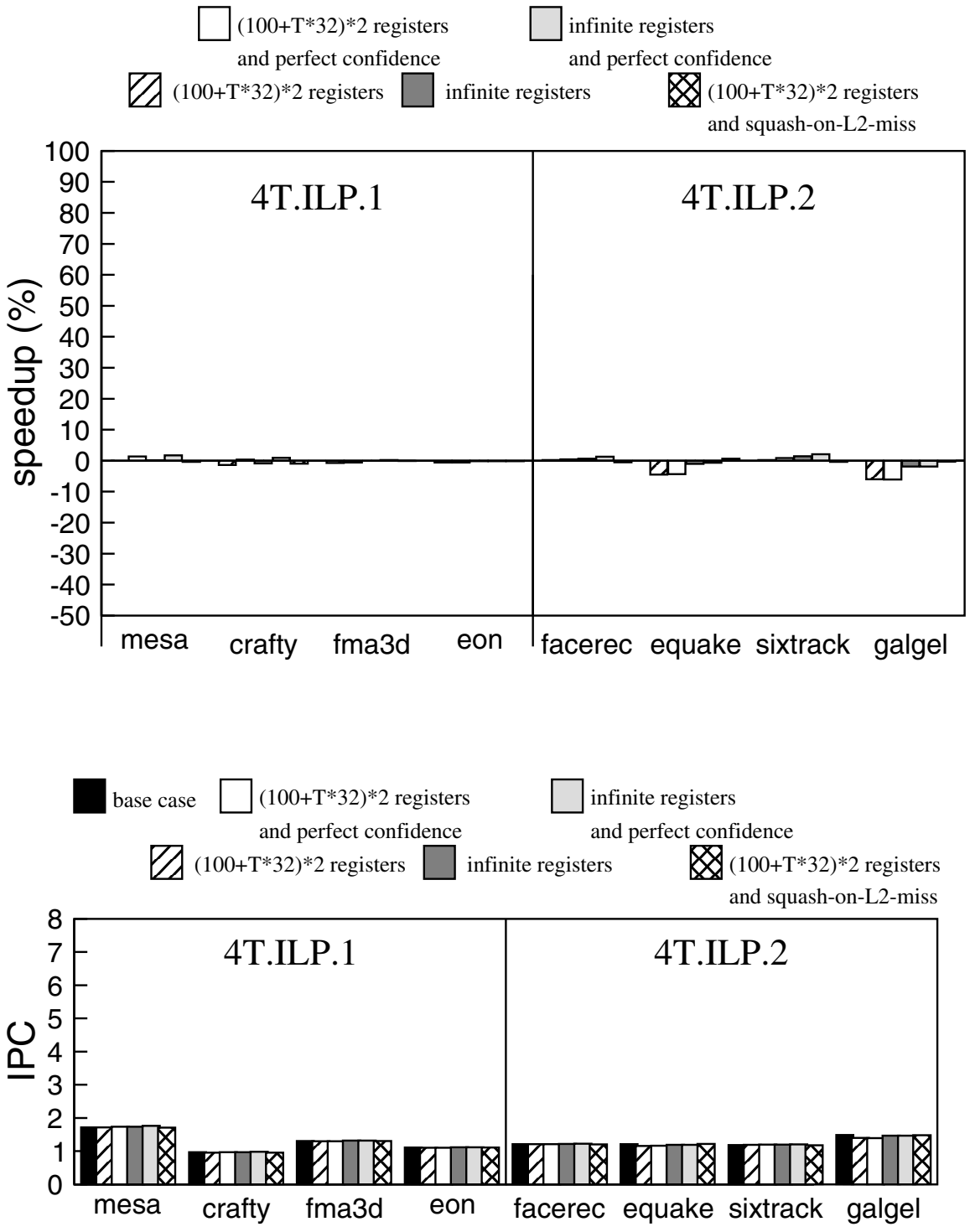


Figure 25: Detailed IPC and speedup of VP for 4T.ILP

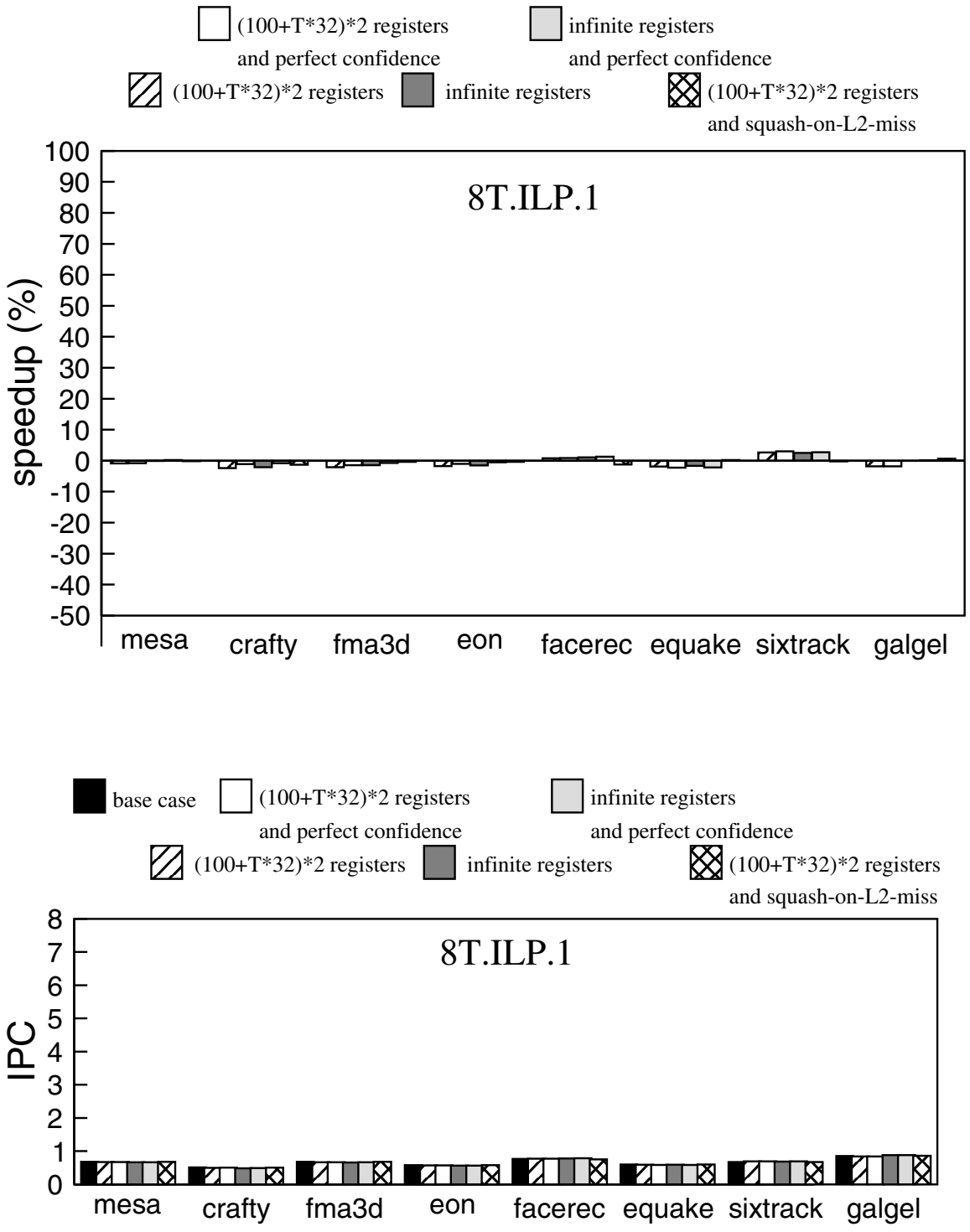


Figure 26: Detailed IPC and speedup of VP for 8T.ILP

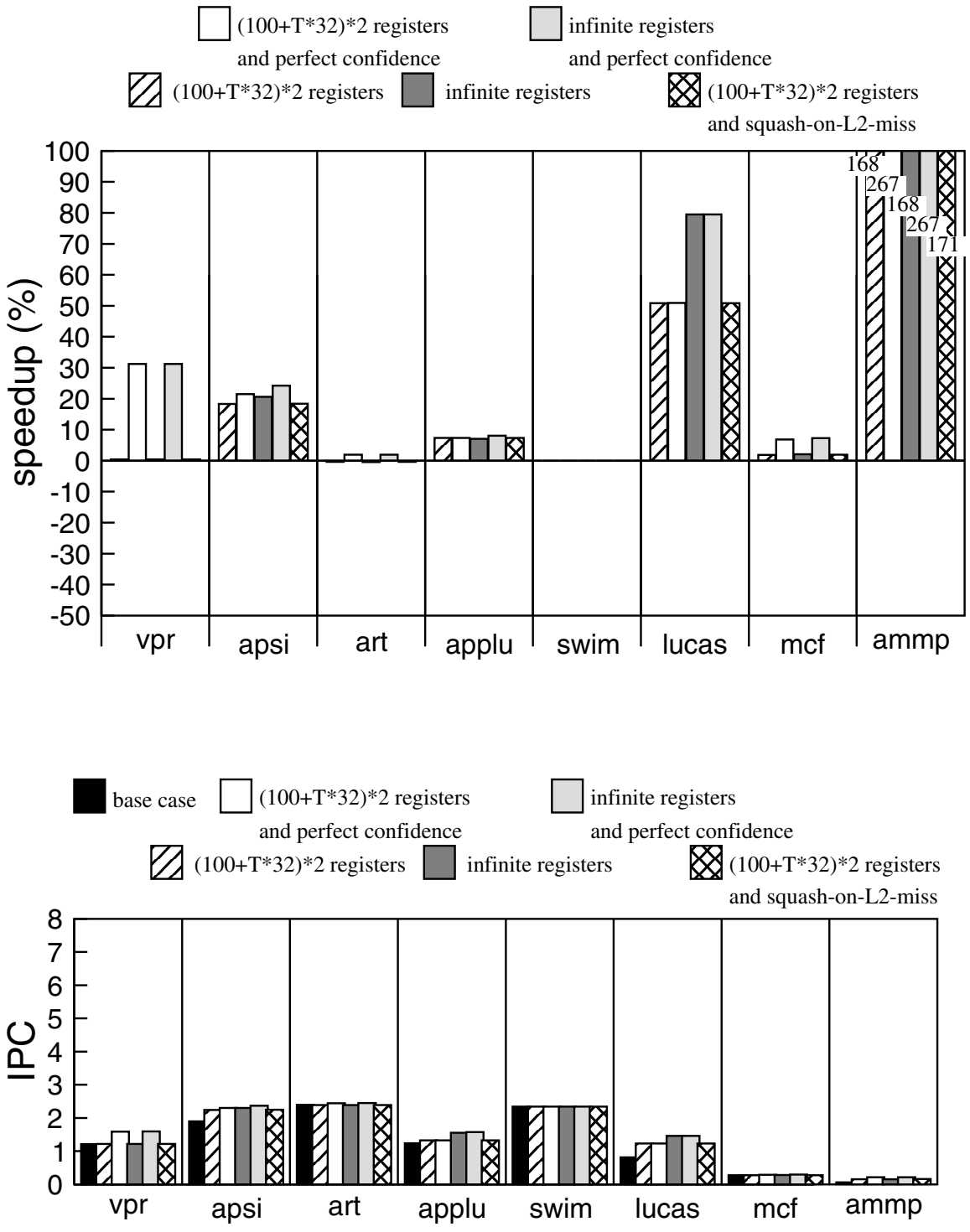


Figure 27: Detailed IPC and speedup of VP for 1T.MEM

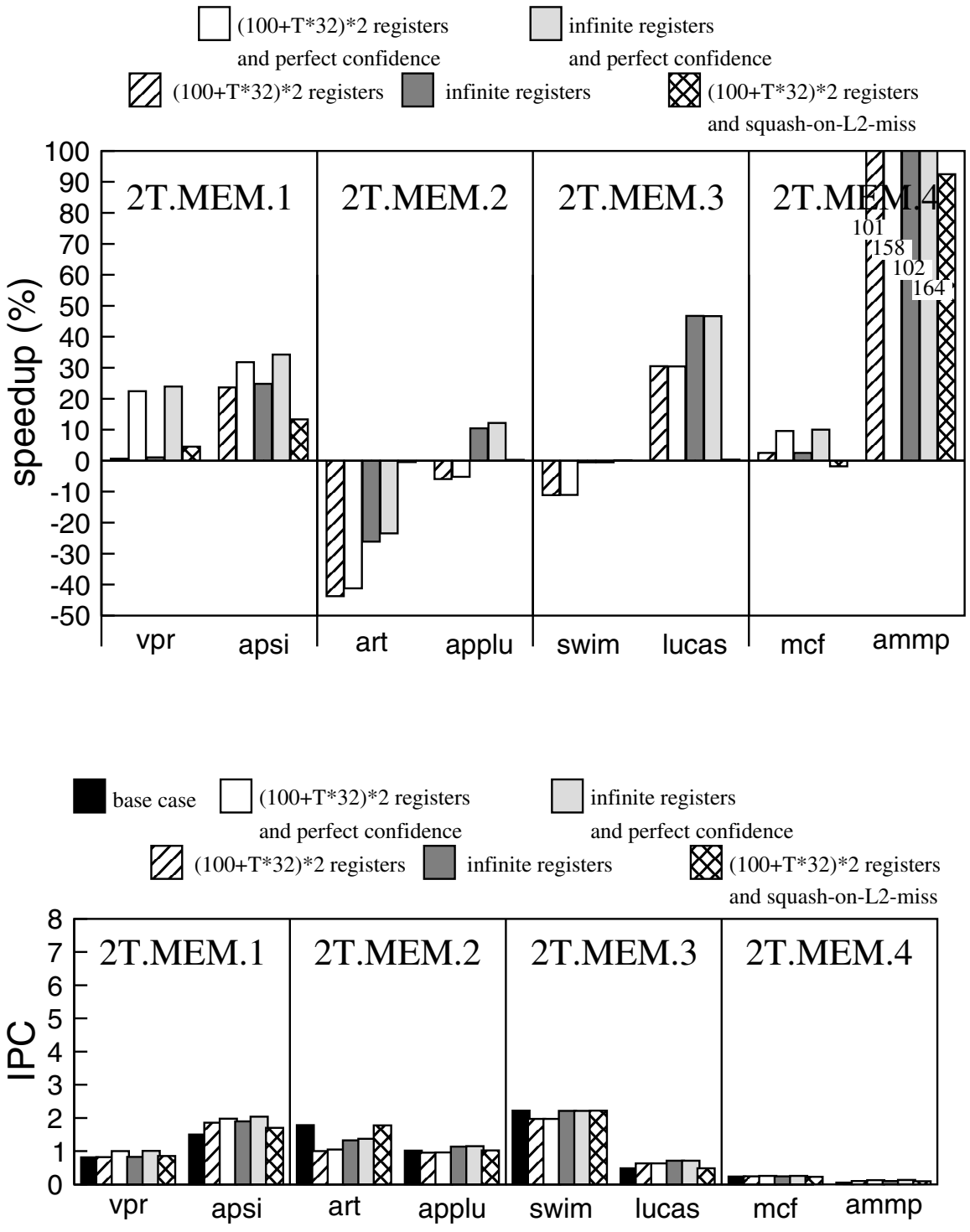


Figure 28: Detailed IPC and speedup of VP for 2T.MEM

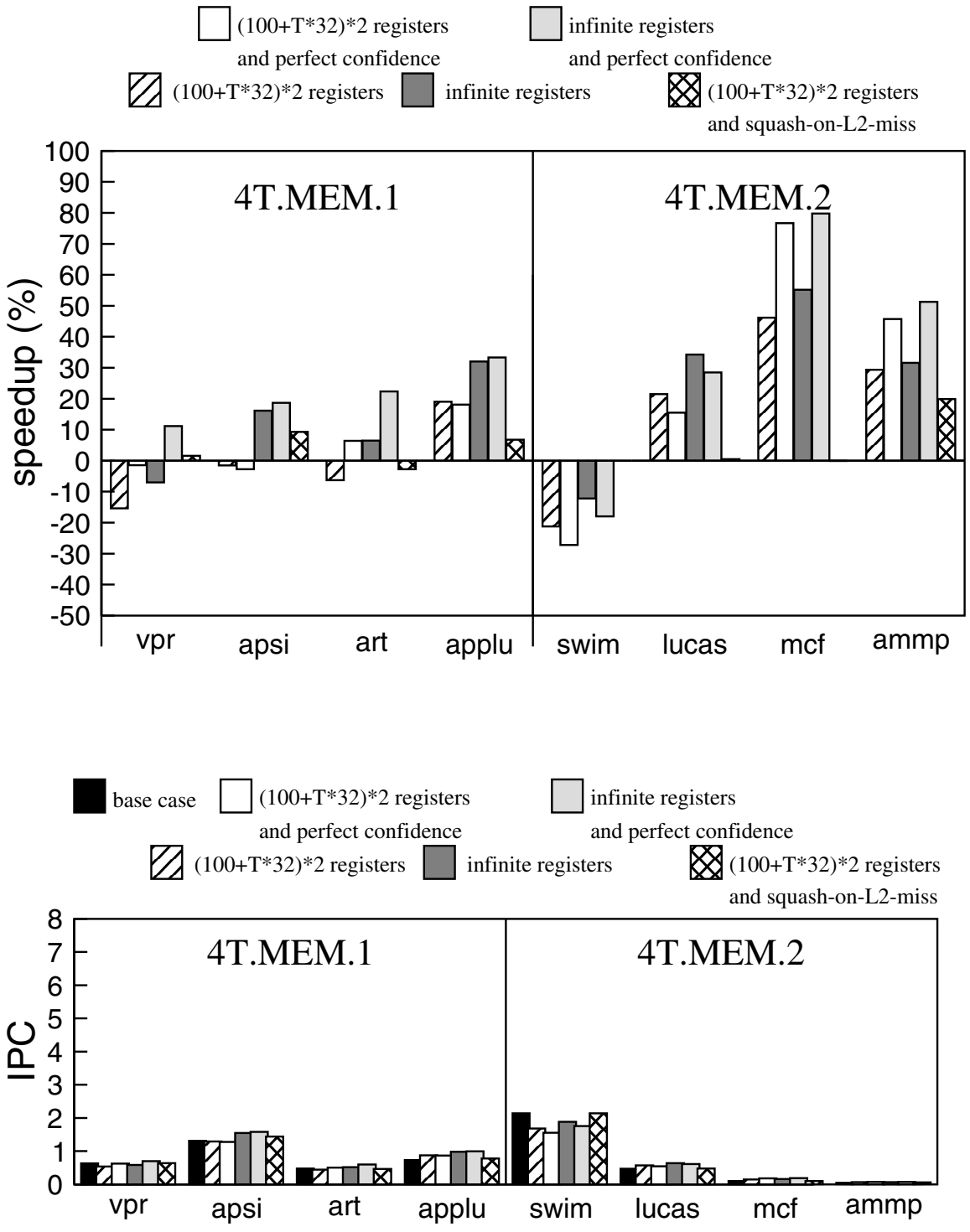
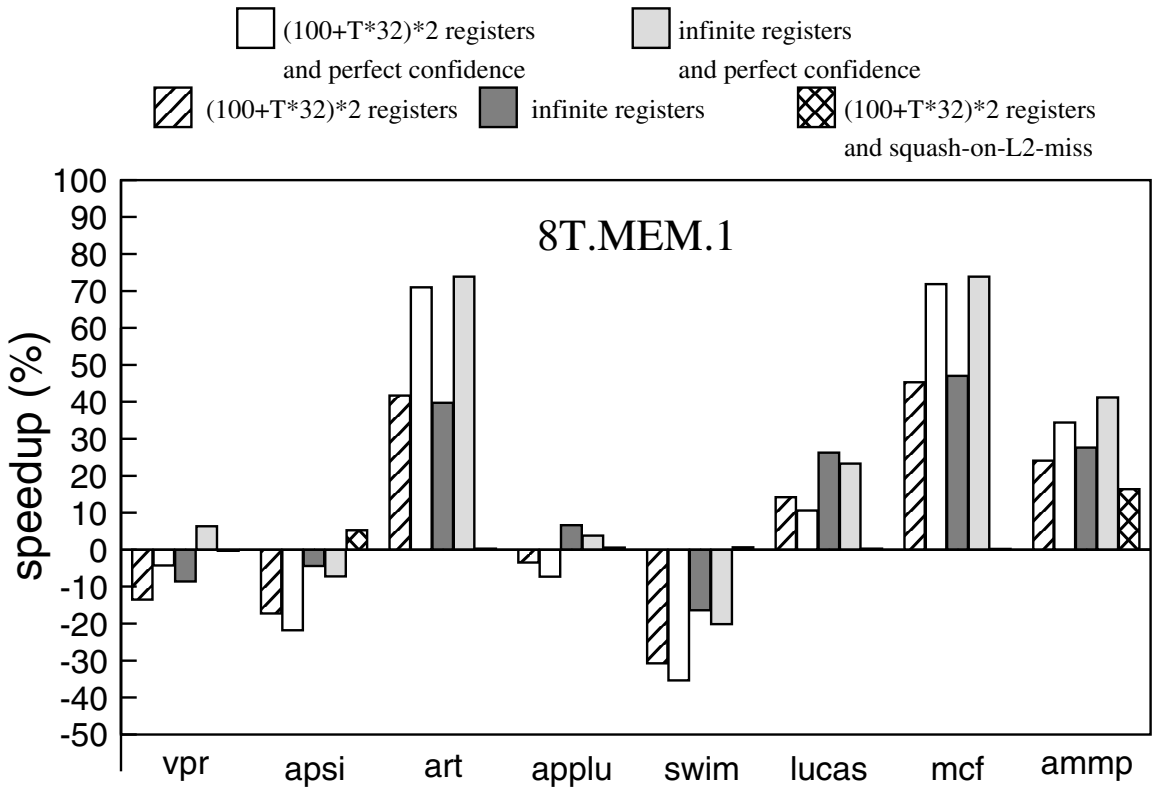


Figure 29: Detailed IPC and speedup of VP for 4T.MEM



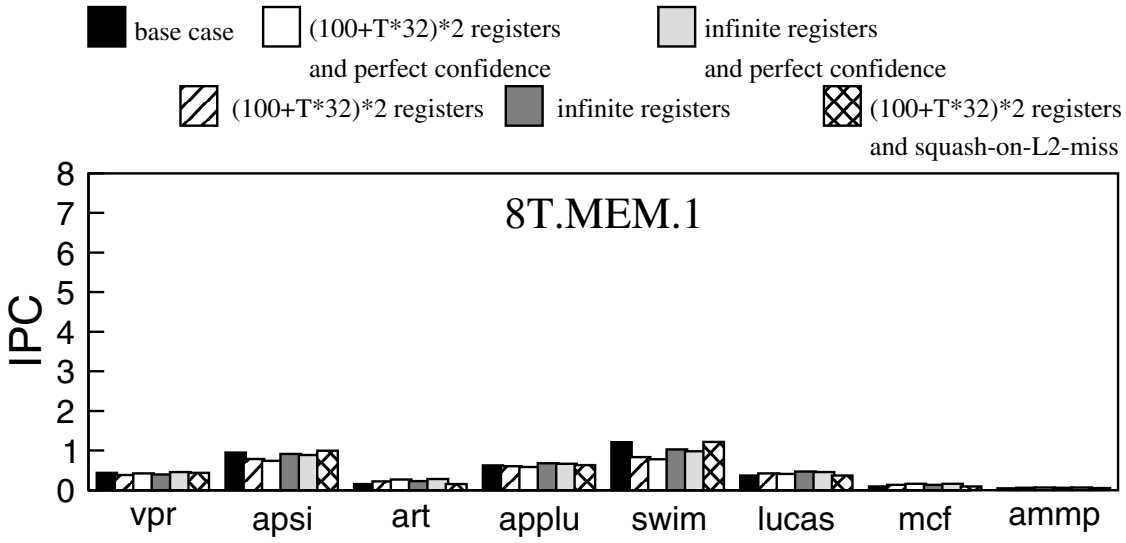


Figure 30: Detailed IPC and speedup of VP for 8T.MEM

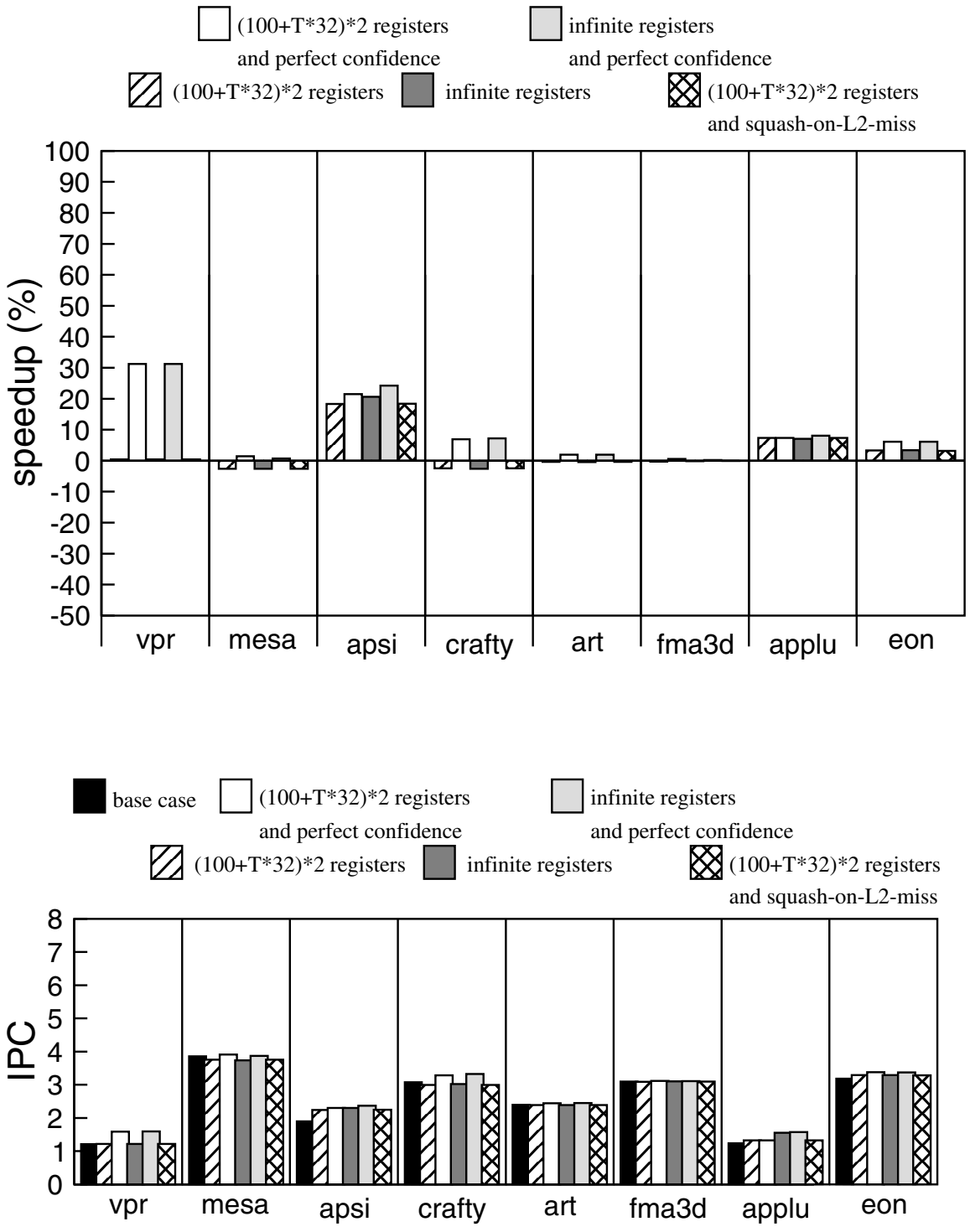


Figure 31: Detailed IPC and speedup of VP for 1T.MIX (part I)

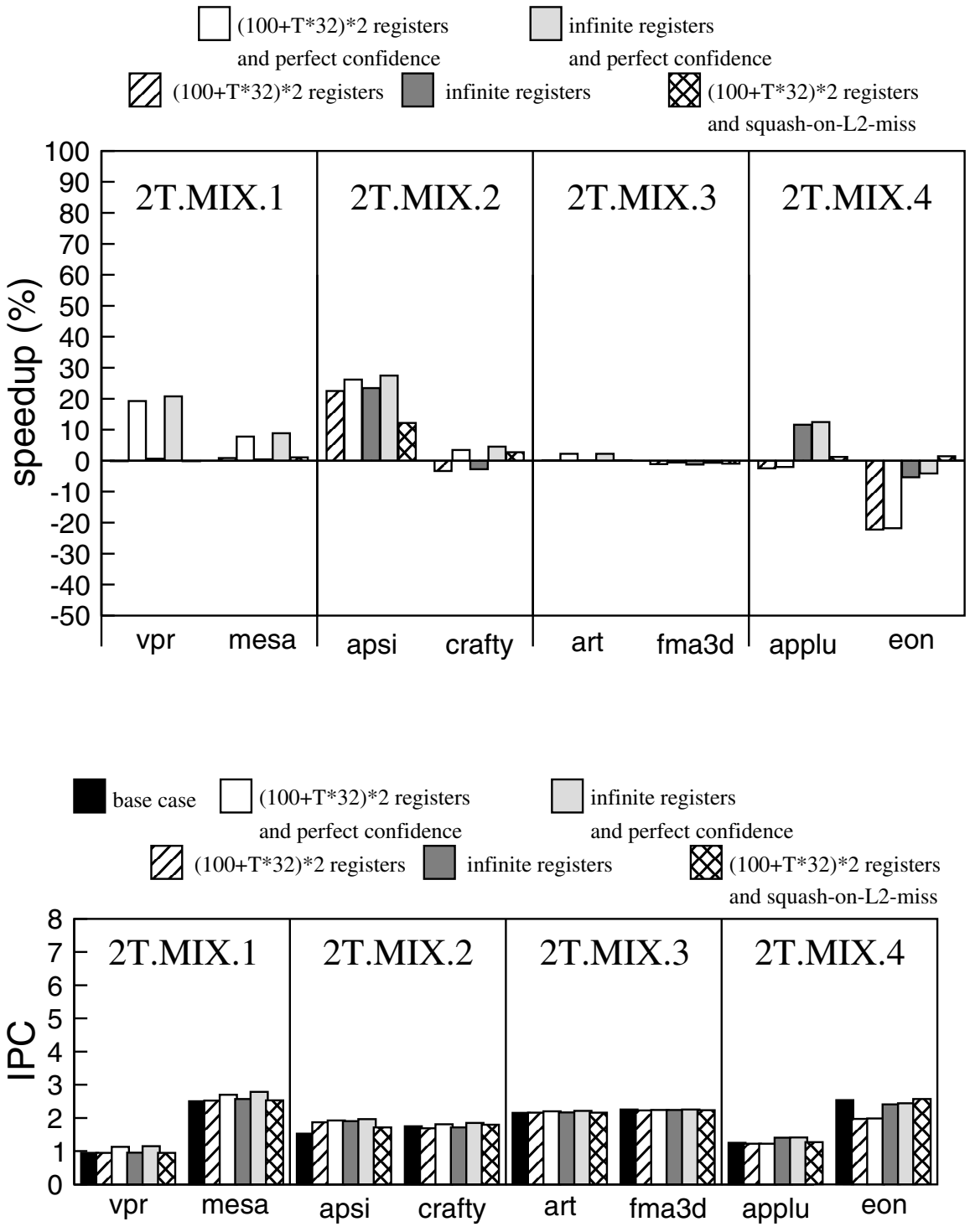


Figure 32: Detailed IPC and speedup of VP for 2T.MIX.{1,2,3,4}

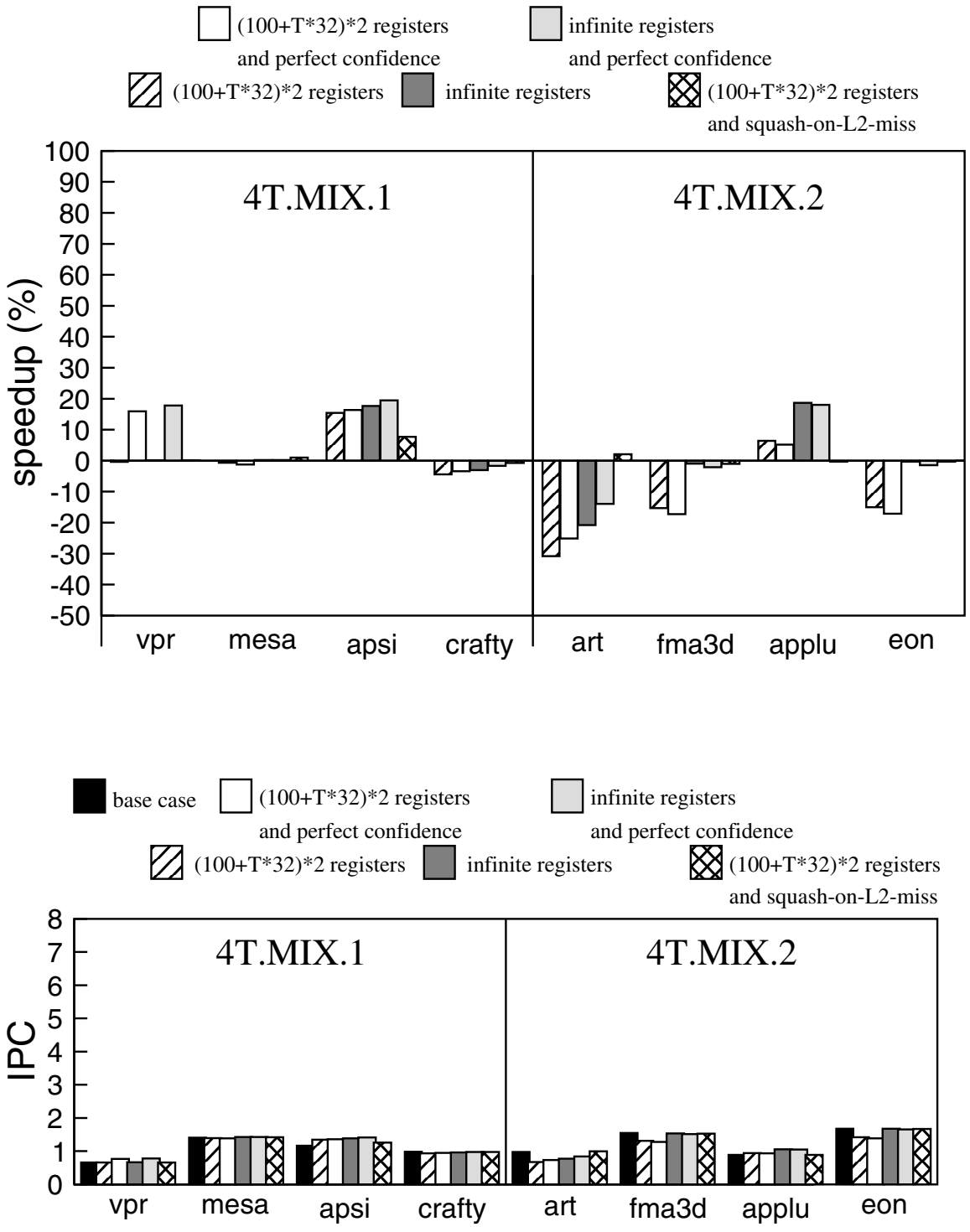


Figure 33: Detailed IPC and speedup of VP for 4T.MIX.{1,2}

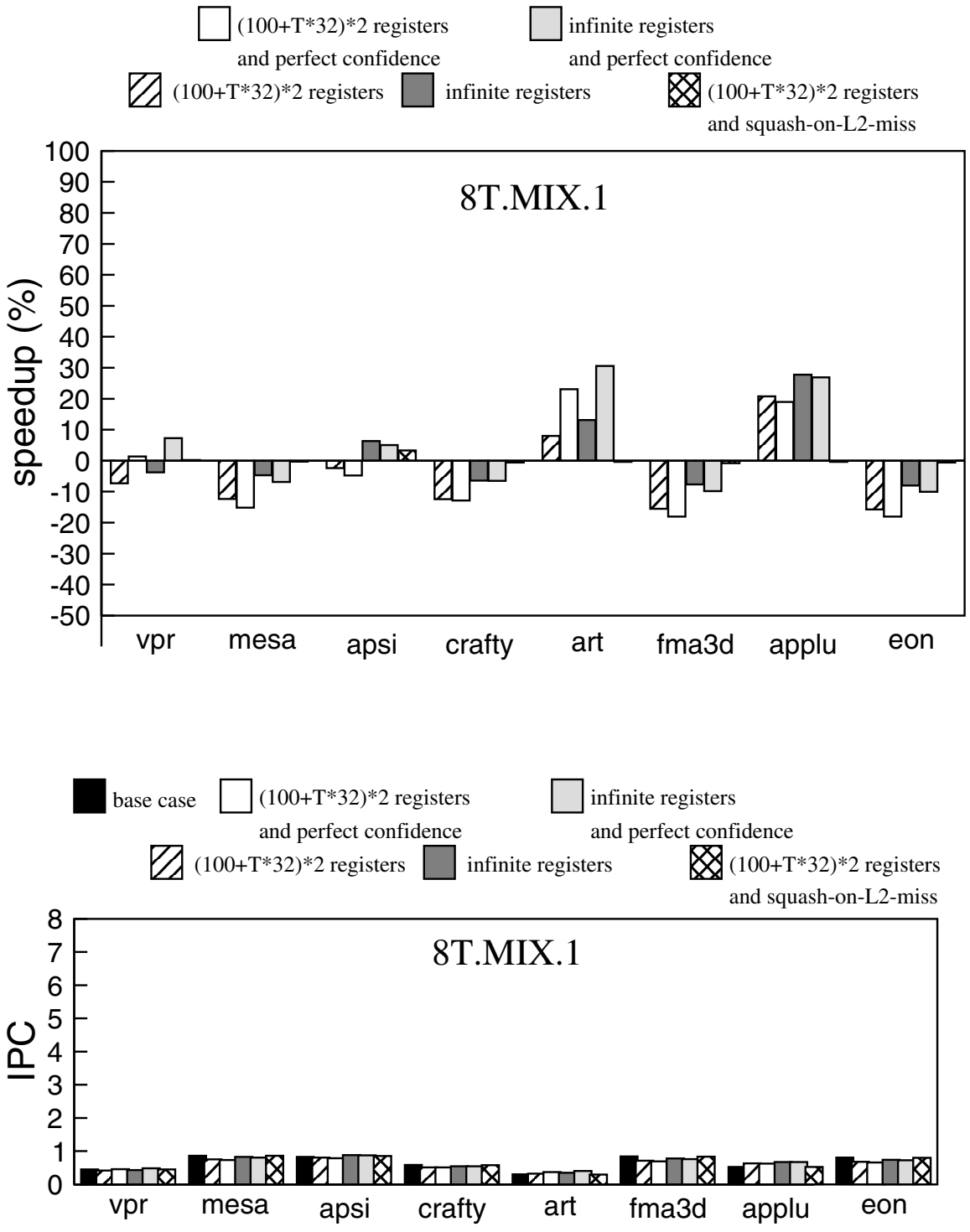


Figure 34: Detailed IPC and speedup of VP for 8T.MIX.1

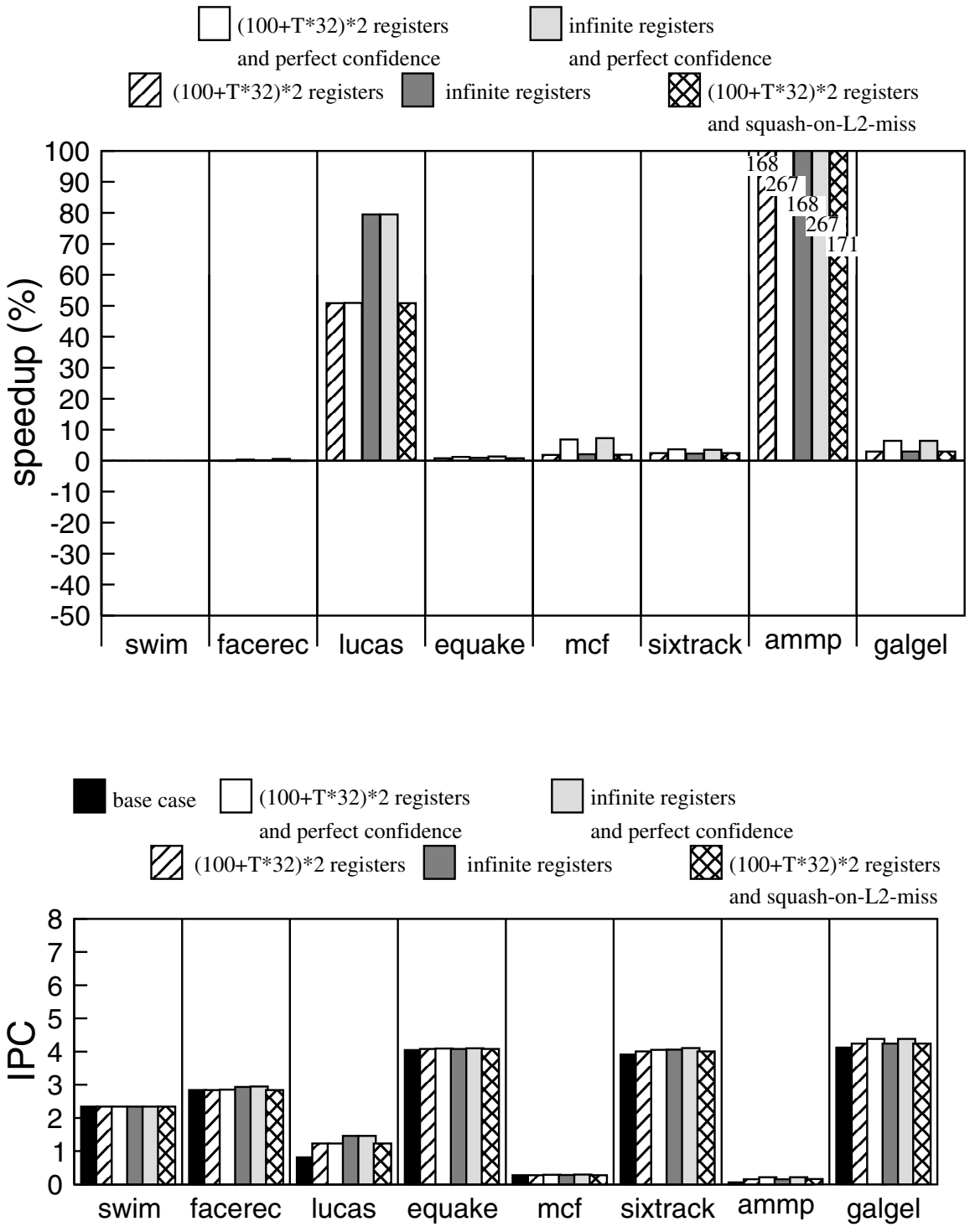


Figure 35: Detailed IPC and speedup of VP for 1T.MIX (part II)

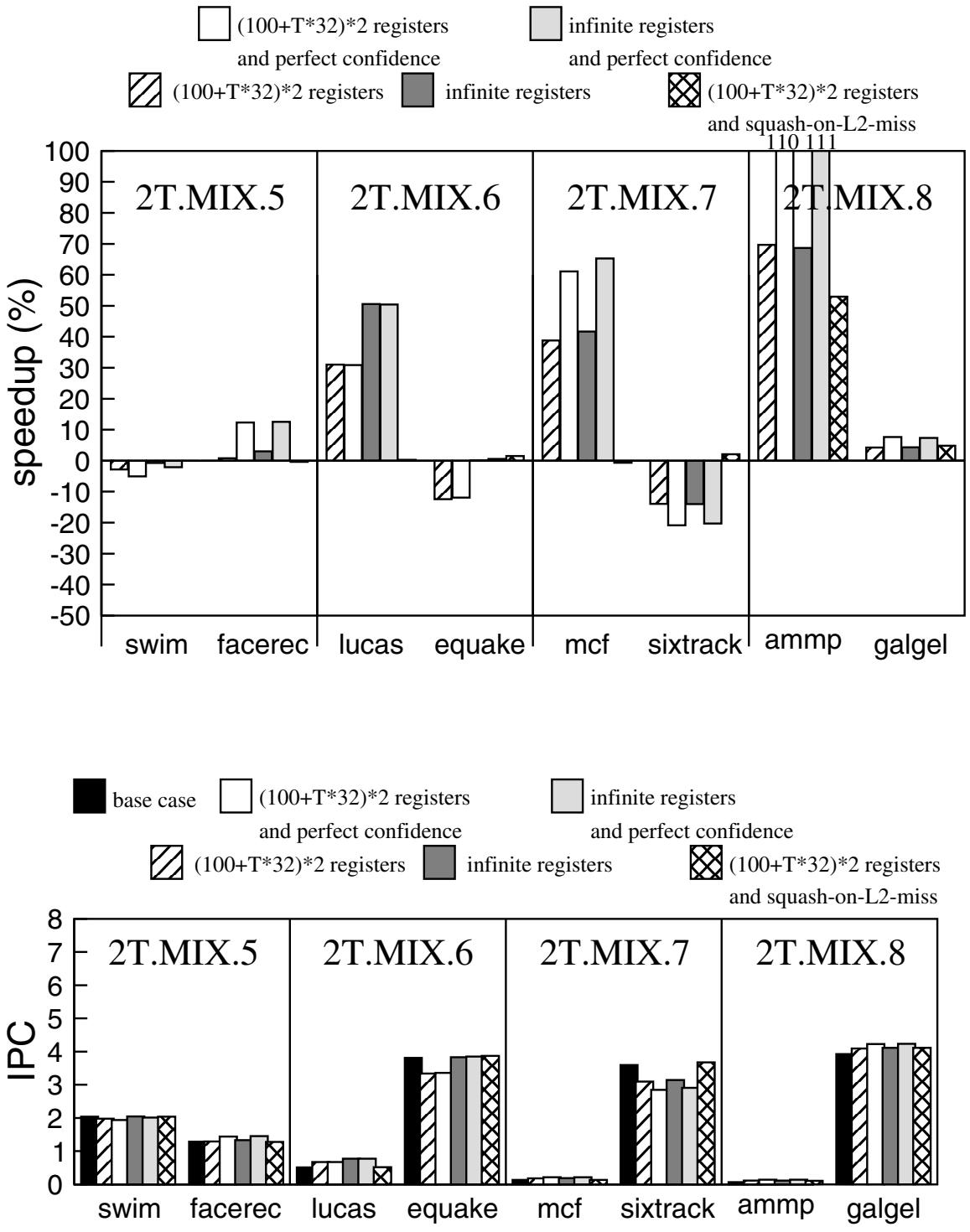


Figure 36: Detailed IPC and speedup of VP for 2T.MIX.{5,6,7,8}

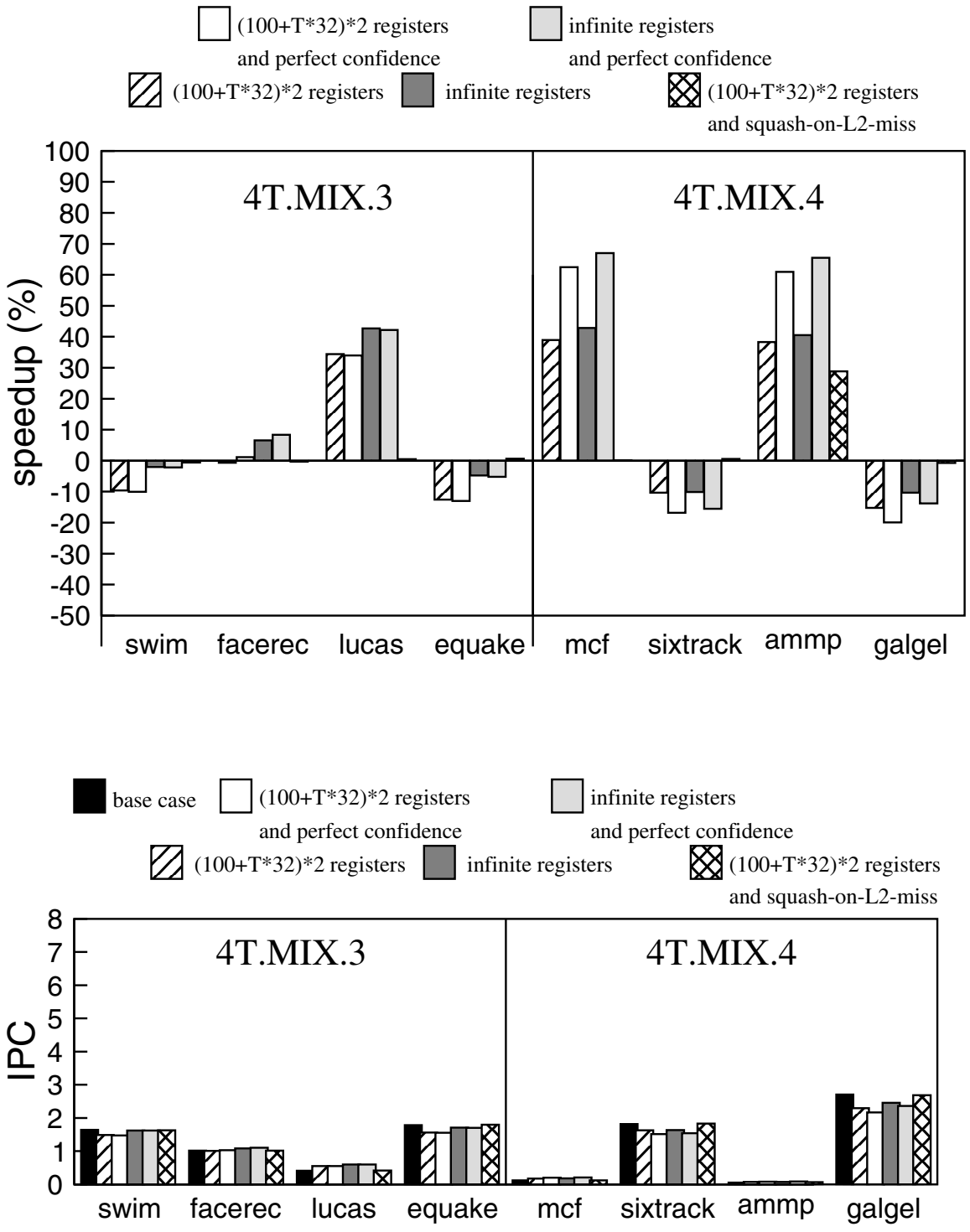


Figure 37: Detailed IPC and speedup of VP for 4T.MIX.{3,4}

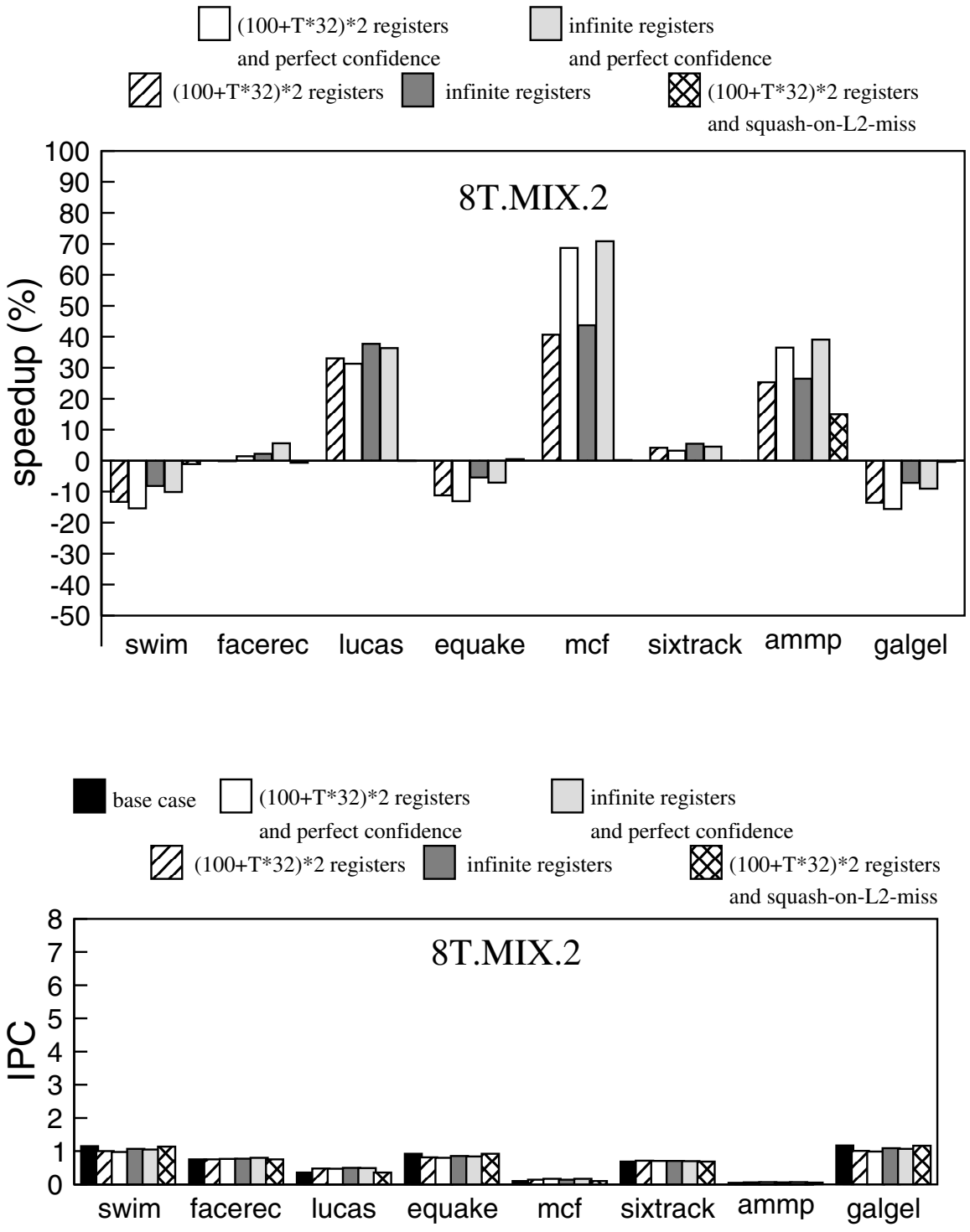


Figure 38: Detailed IPC and speedup of VP for 8T.MIX.2

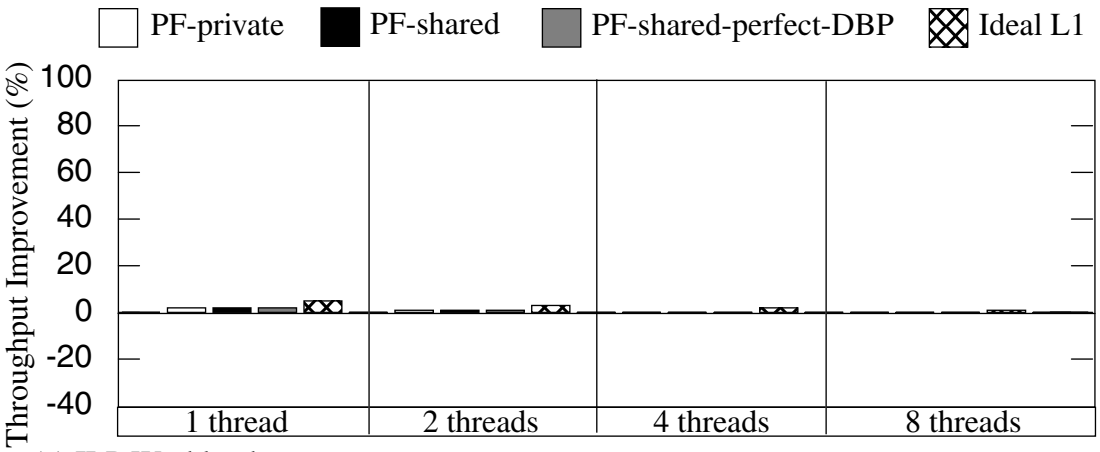
5.3 Prefetching Results

Recall from Section 1 that because SMT tolerates latency but at the same time increases pressure on the memory hierarchy by overlapping multiple threads, the opportunity for PF is unknown. While PF reduces memory latency, prefetching into L2 encourages L1 misses in fewer cycles, which causes clogging of the issue queue and slows down the other threads. Ironically, only correct prefetches cause this clogging. Thus, SMT's sharing of the issue queue across multiple threads impacts PF's effectiveness. In this section, I evaluate these opposing effects of PF in SMT.

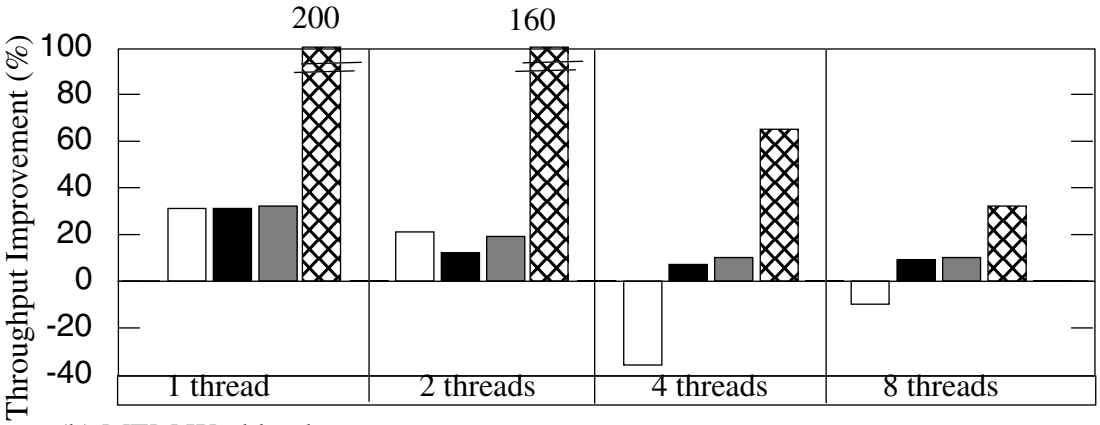
While I showed that TC and VP do not improve SMT throughput; in this section, I will show that PF improves throughput for MEM, but has limited opportunity for MIX.

Figure 39 shows PF's throughput improvements compared to the SMT without PF. As before, Figure 39 shows the three sets of workloads, ILP, MEM, and MIX, and varies the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set. Note that the Y-axis scale has changed from the previous graphs.

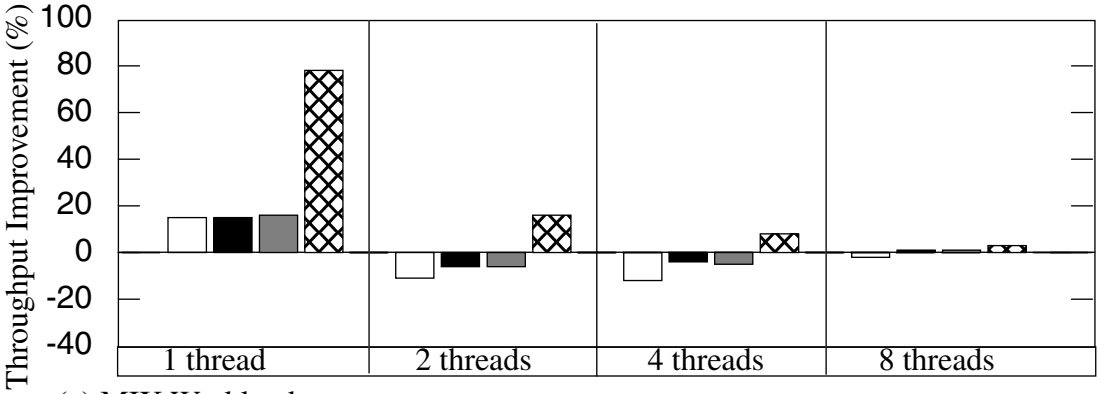
Because prefetching into L2 causes the issue queue clogging problem, I studied ways to reduce this clogging. First, I experimented with prefetching into L1. Unlike prefetching into L2, prefetching into L1 cannot use L1 misses as triggers (prefetched block will displace useful data) and needs dead-block prediction [55]. I found that because of high pressure on L1 in SMT, the dead time is shorter in SMT than that in a single thread. The shorter dead time makes dead block prediction harder. Second, I resumed prefetching into L2 and tried to avoid clogging by preventing instructions which are past an L1 miss that hits in a prefetched L2 block from entering the pipeline. To this end, I used an L1 miss predictor which stops fetching past a predicted L1 miss. The predictor essentially needs to balance accuracy (avoid incorrectly stopping fetch due to mispredictions) and coverage (identify all the misses). Unfortunately, achieving this balance proved to be difficult. Therefore, I looked into other ways to reduce the clogging. Taking a hint from VP, which reduces coverage to reduce mispredictions, I tried to reduce prefetch coverage to prevent



(a) ILP Workload.



(b) MEM Workload.



(c) MIX Workload.

Figure 39: Prefetching throughput improvements.

too many L2 misses from being converted to L1 misses, which cause clogging. Reasoning that a shared first-level history would have less coverage than a private first-level history due to inter-thread interference, I experimented with these two configurations. I found that the shared first-level history works better and achieves throughput improvements. Because PF already shows throughput improvement for MEM and little opportunity for MIX, further enhancements through either prefetching into L1 or using a better L1 miss predictor will improve MEM's throughput more and will not improve MIX. Any such further enhancement will only reinforce my conclusions.

Figure 39 shows a group of four bars for a given number of threads. The first bar shows *PF-private*, which uses a private first-level history. The second bar shows *PF-shared*, which uses a shared first-level history. To confirm that the absence of dead-block prediction does not affect PF (as previously shown in [16]), the third bar, *PF-shared-prefect-DBP,* shows PF with perfect L2-dead-block prediction. To show the potential of PF, the last bar shows *Ideal L1*, which lets every access from the processor hit in L1.

From Figure 39, we see that PF does not benefit ILP much with multiple threads, because ILP has low L1 miss rates. Looking at MEM, PF significantly improves single-thread performance. This result agrees with the results from previous PF papers [55,16], indicating that my prefetcher is implemented correctly. With multiple threads, we see that while *PF-private* degrades throughput, *PF-shared* improves throughput. *PF-private's* poor performance is due to issue queue clogging, as can be seen in the second and third rows in Table 8 by the larger fraction of time the issue queue stays clogged (*issueQ clog frac.*) with *PF-private* than with *PF-shared*. Note that the *base case throughput*, shown in the first row of Table 8, continues to improve as I increase the number of threads to eight, showing that *PF-private's* diminishing returns are *not* due to pipeline saturation. Thus we see that SMT's sharing of the issue queue among its threads accounts for the difference between *PF-private's* failure and *PF-shared's* success.

We can also see that *PF-shared-perfect-DBP* is marginally better than *PF-shared*, showing that using L1 misses as triggers is a good dead-block predictor, as also claimed by [16]. *Ideal L1* shows that though the opportunity reduces with more threads, there is

still substantial opportunity even with eight threads. *PF-shared* captures some of this opportunity, achieving 7% improvement with four threads and 9% with eight threads. These results show that when memory latency is a major bottleneck, even multiple threads cannot tolerate all L2 misses, and PF is effective.

For MIX, *PF-shared* suffers 6% and 4% degradation with two and four threads, respectively. Despite using a shared configuration, this workload causes issue queue clogging, resulting in slight throughput degradation. This result is not surprising when we look at *Ideal L1*, which shows little opportunity for PF with increasing threads. This limited opportunity combined with issue queue clogging forces *PF-shared* to degrade with two or more threads.

Table 8 presents important statistics for PF. The fourth, fifth, and sixth rows show the *L2 miss rates* in the base case and *PF-private* and *PF-shared*, respectively. These miss rates confirm that PF is effective in reducing L2 misses. *PF-private's* miss rates are lower than those of *PF-shared*, indicating that *PF-private* has higher coverage than *PF-shared*. This higher coverage causes clogging problems that result in throughput degradation. The next two rows show that the *accuracy* of *PF-private* and *PF-shared* behave similarly to coverage and have the same effect. Finally, I show *weighted speedup* for *PF-private* and *PF-shared*. *PF-private* has positive weighted speedups for MEM and MIX while it degrades throughput, showing that *PF-private* improves low-IPC threads with high miss rates at the cost of overall throughput. *PF-shared* has positive weighted speedups but lower than those of *PF-private* due to lower coverage. For MEM, *PF-shared* has both positive weighted speedups and improved throughput, indicating that *PF-shared* improves low-IPC threads without hurting the other threads.

Similar to previous section, because some of my experiments show negative results for PF, I ensure that the negative results are not caused by the utilization for the SMT base case that is overly high such that it leaves no headroom for PF to improve. To that end, , I measure the utilizations for the eight-issue SMT base case. The utilization is defined as the number of instructions fetched, regardless of whether it is committed or squashed later in the pipeline, divided by the execution time. The utilizations for the eight-issue SMT base

Table 8: Prefetching statistics

	ILP workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	3.5	4.6	5.1	5.3
PF-private issueQ clog frac. (%)	2.5	1.0	0.3	0.3
PF-shared issueQ clog frac. (%)	2.5	1.0	0.5	0.4
Base case L2 miss rate (%)	9.8	7.5	9.1	4.8
PF-private L2 miss rate (%)	7.9	5.0	4.9	3.4
PF-shared L2 miss rate (%)	7.9	5.0	7.5	4.6
PF-private Accuracy (%)	22.7	24.6	46.6	68.2
PF-shared Accuracy (%)	22.7	24.0	34.7	30.5
PF-private Weighted Speedup (%)	2.0	1.2	0.5	0.3
PF-shared Weighted Speedup (%)	2.0	1.0	0.2	0.0
	MEM workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	1.2	1.8	3.1	4.0
PF-private issueQ clog frac. (%)	37.7	25.2	36.8	18.2
PF-shared issueQ clog frac. (%)	37.7	24.8	11.5	6.1
Base case L2 miss rate (%)	26.8	25.3	26.7	32.6
PF-private L2 miss rate (%)	12.2	9.1	12.0	15.8
PF-shared L2 miss rate (%)	12.2	16.9	19.9	26.5
PF-private Accuracy (%)	66.1	85.1	83.7	86.9
PF-shared Accuracy (%)	66.1	73.6	67.6	62.2
PF-private Weighted Speedup (%)	30.7	53.0	22.3	20.9
PF-shared Weighted Speedup (%)	30.7	21.7	14.7	10.7
	MIX workload			
	1 thread	2 threads	4 threads	8 threads
Base case throughput (IPC)	2.1	3.8	4.7	5.2
PF-private issueQ clog frac. (%)	18.8	10.3	7.2	1.7
PF-shared issueQ clog frac. (%)	18.8	9.6	3.9	1.1
Base case L2 miss rate (%)	18.0	24.8	21.9	21.9
PF-private L2 miss rate (%)	10.0	11.1	9.5	10.8
PF-shared L2 miss rate (%)	10.0	12.4	17.1	18.6
PF-private Accuracy (%)	42.7	63.2	81.3	80.5
PF-shared Accuracy (%)	42.7	59.3	66.5	55.9
PF-private Weighted Speedup (%)	15.5	19.7	26.0	12.4
PF-shared Weighted Speedup (%)	15.5	16.7	7.2	2.4

case used in the PF experiments with eight threads are 5.6, 4.4 and 5.5 for ILP, MEM and MIX workloads, recpectively.

My experiments do not give PF any undue advantage and yet show that PF improves SMT throughput for MEM. Because PF hides L2-miss latencies, using longer latencies will further improve throughput. For MIX workload, I showed that PF does not improve even with an ideal L1. Therefore, my results unequivocally prove that PF improves MEM and does not improve MIX, and there is no need to vary other parameters.

Figure 40 through Figure 55 shows PF's speedup and IPC for each workload, as well as the speedup and IPC components in the multi-programmed workloads. These graphs were summarized and shown in Figure 39. In each figure, the top graph shows the speedup of each thread in a workload and the bottom graph shows the IPC of each thread in a workload. The sum of IPCs of these thread in a workload forms the total throughput of the workload.

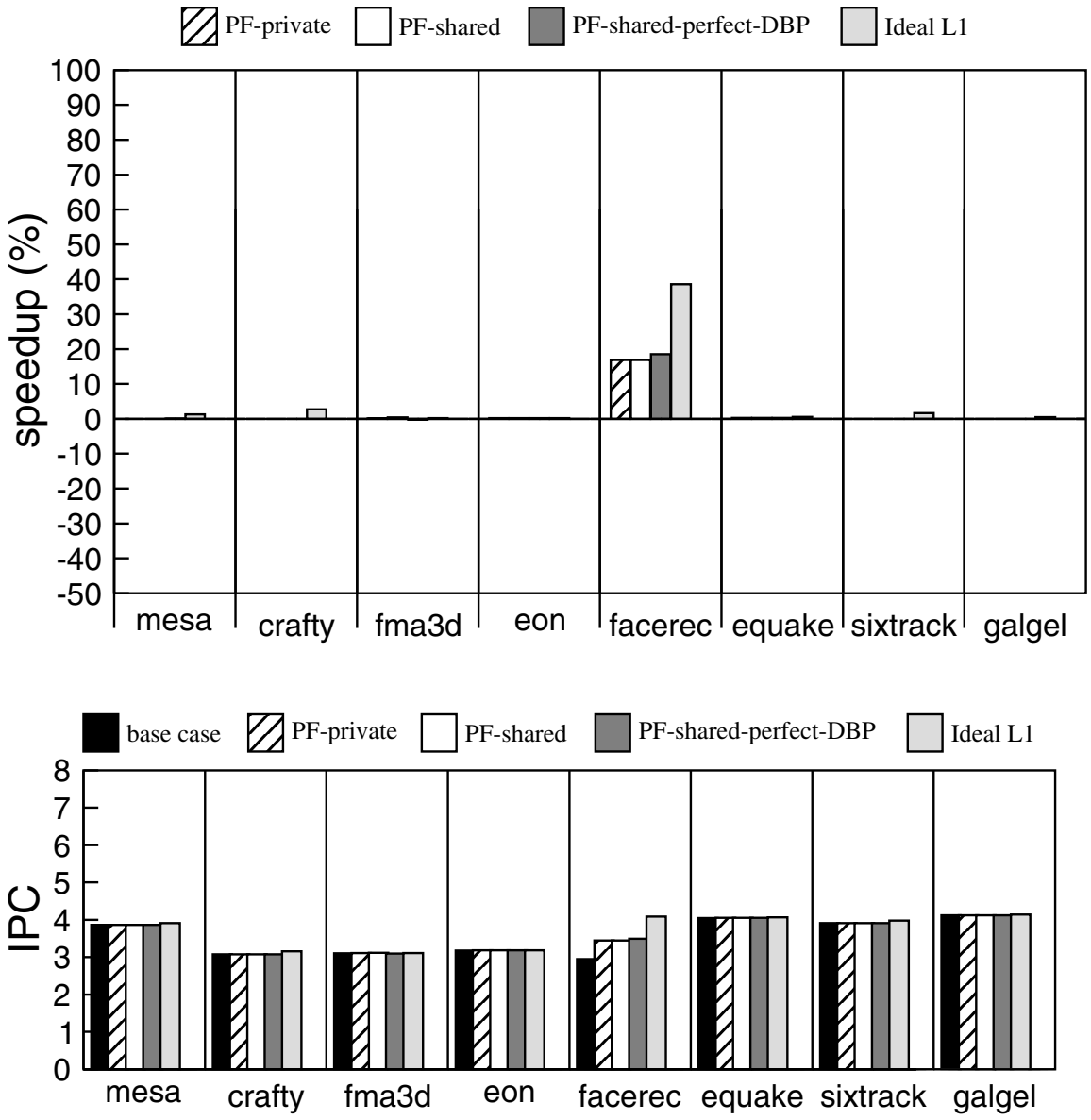


Figure 40: Detailed IPC and speedup of PF for 1T.ILP

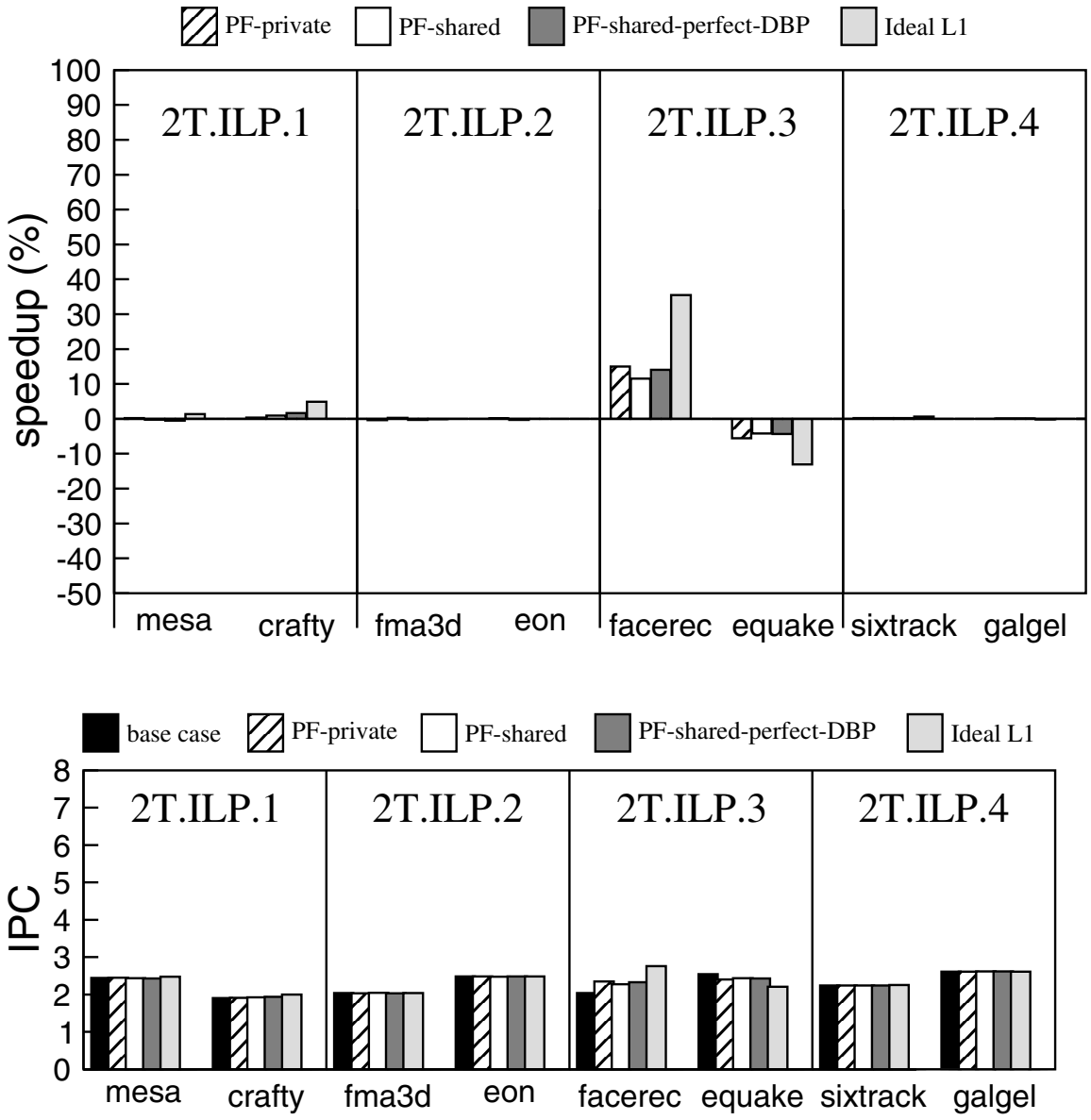


Figure 41: Detailed IPC and speedup of PF for 2T.ILP

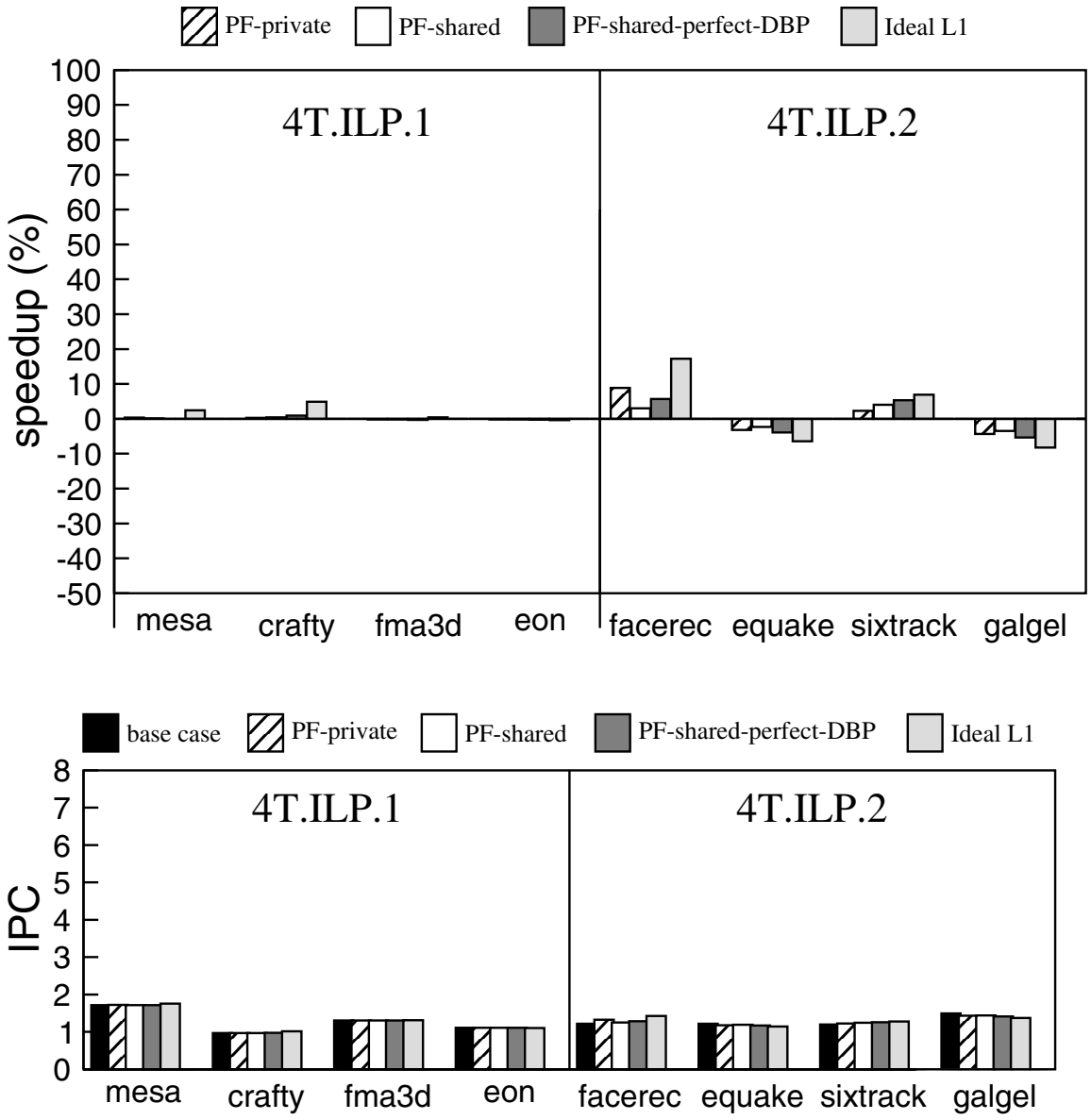


Figure 42: Detailed IPC and speedup of PF for 4T.ILP

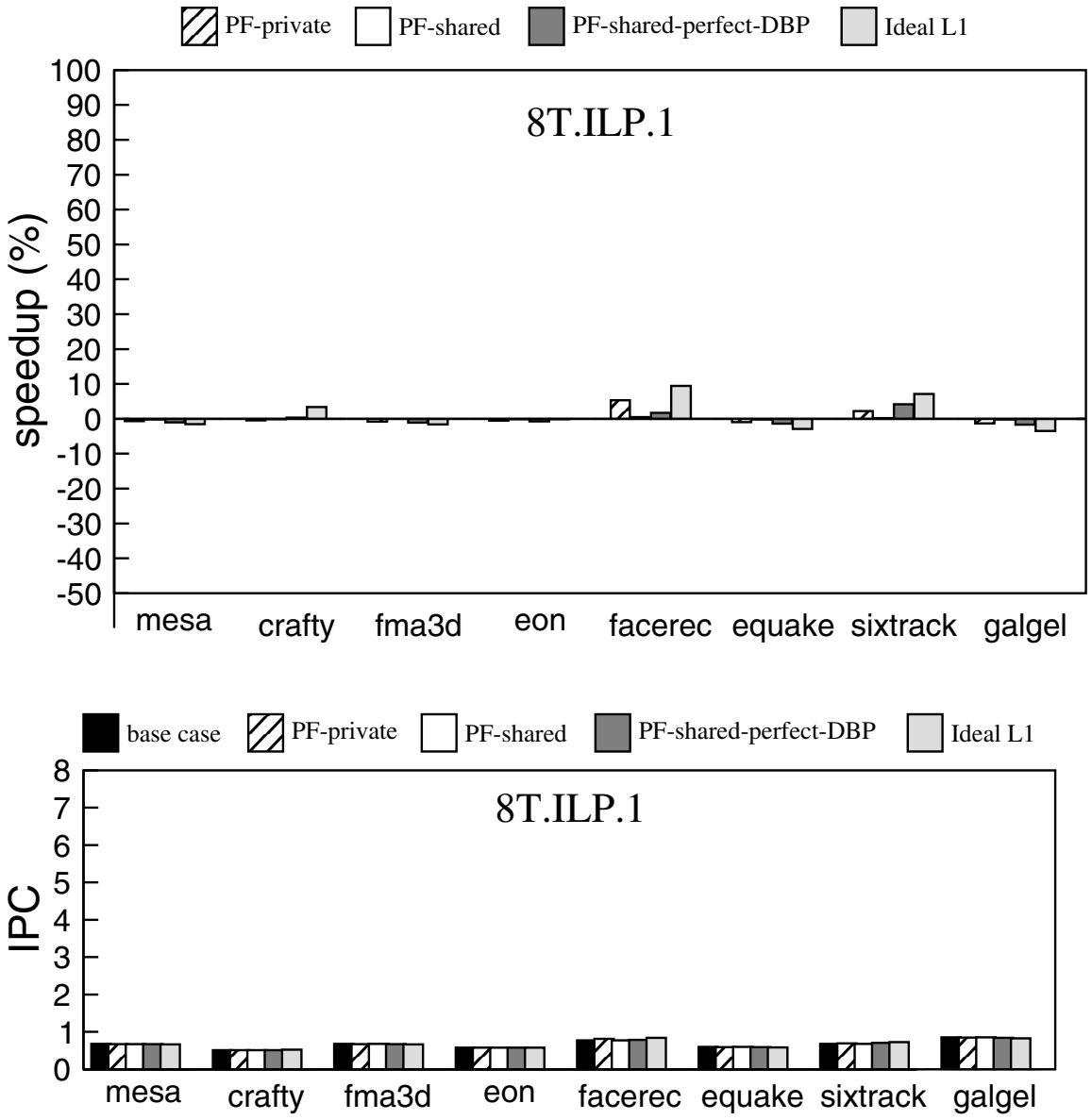


Figure 43: Detailed IPC and speedup of PF for 8T.ILP

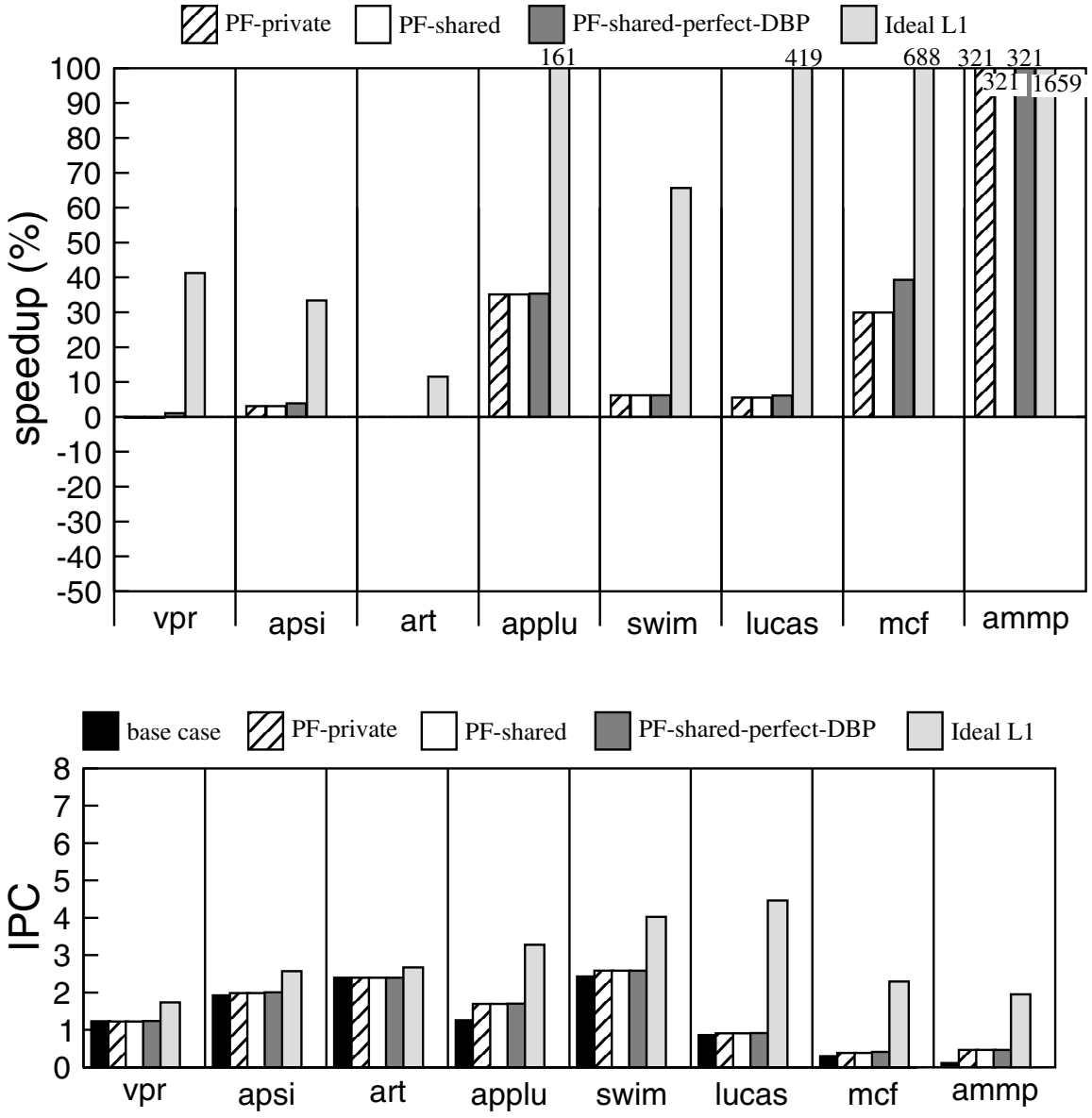


Figure 44: Detailed IPC and speedup of PF for 1T.MEM

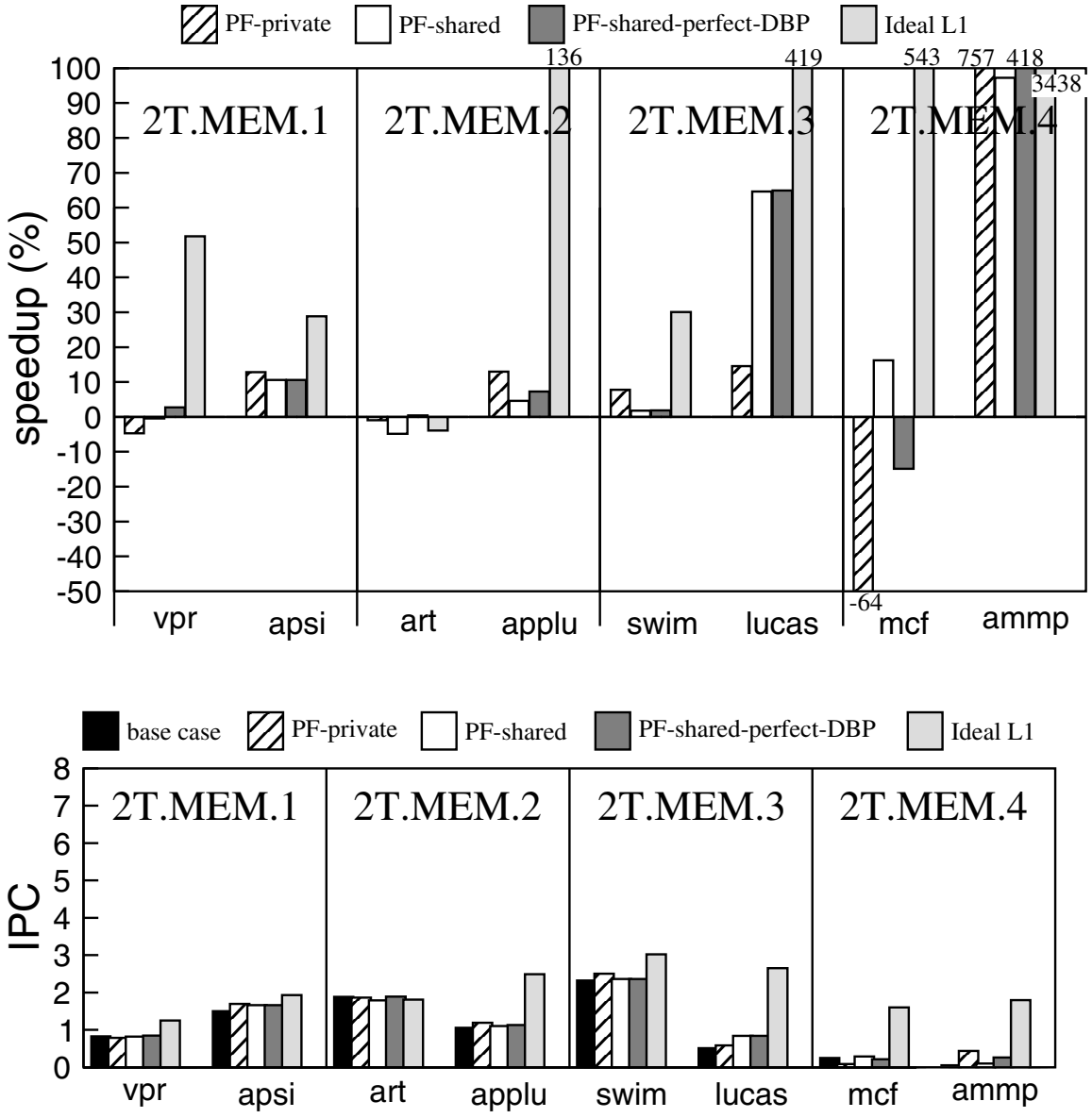


Figure 45: Detailed IPC and speedup of PF for 2T.MEM

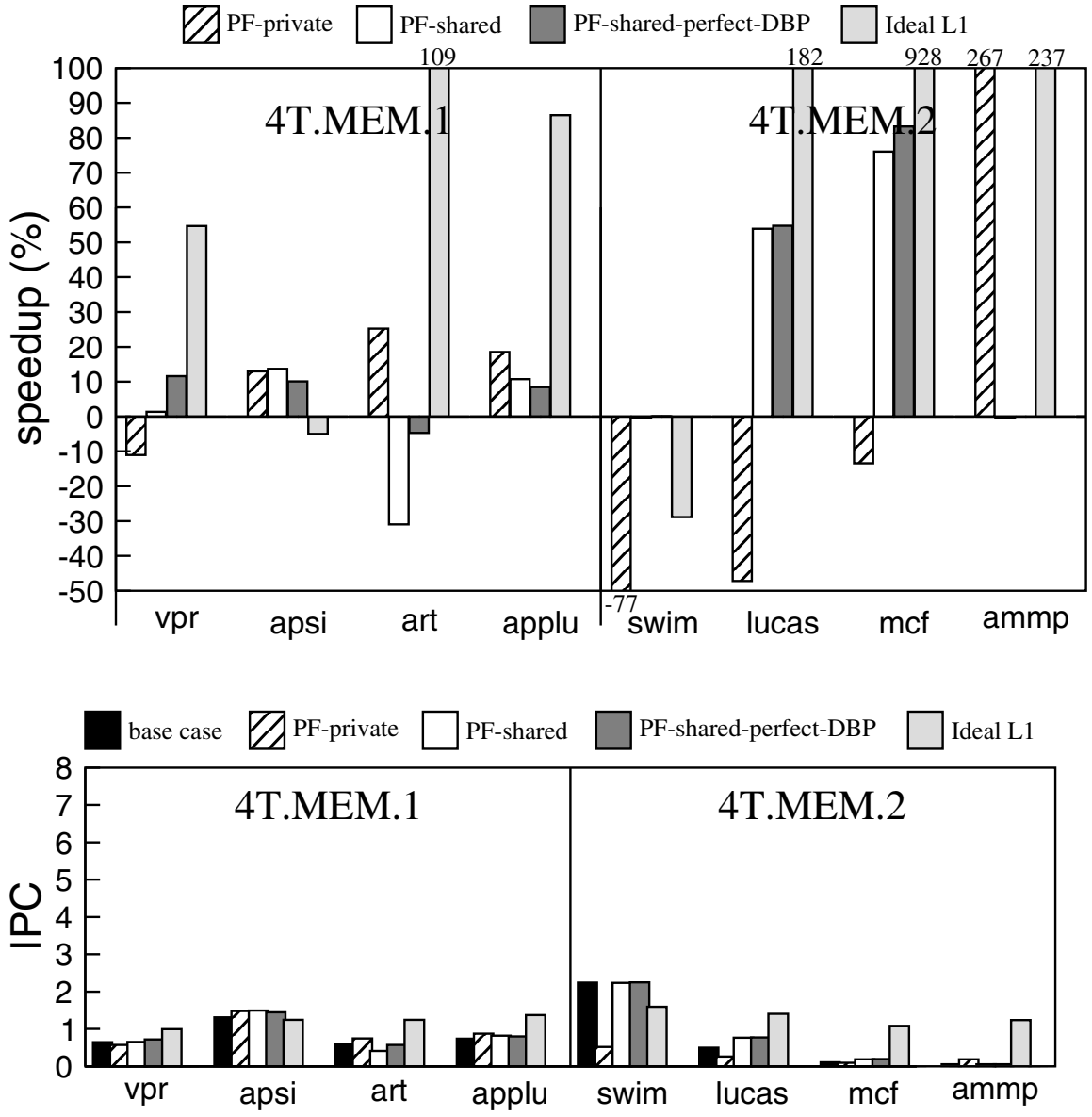


Figure 46: Detailed IPC and speedup of PF for 4T.MEM

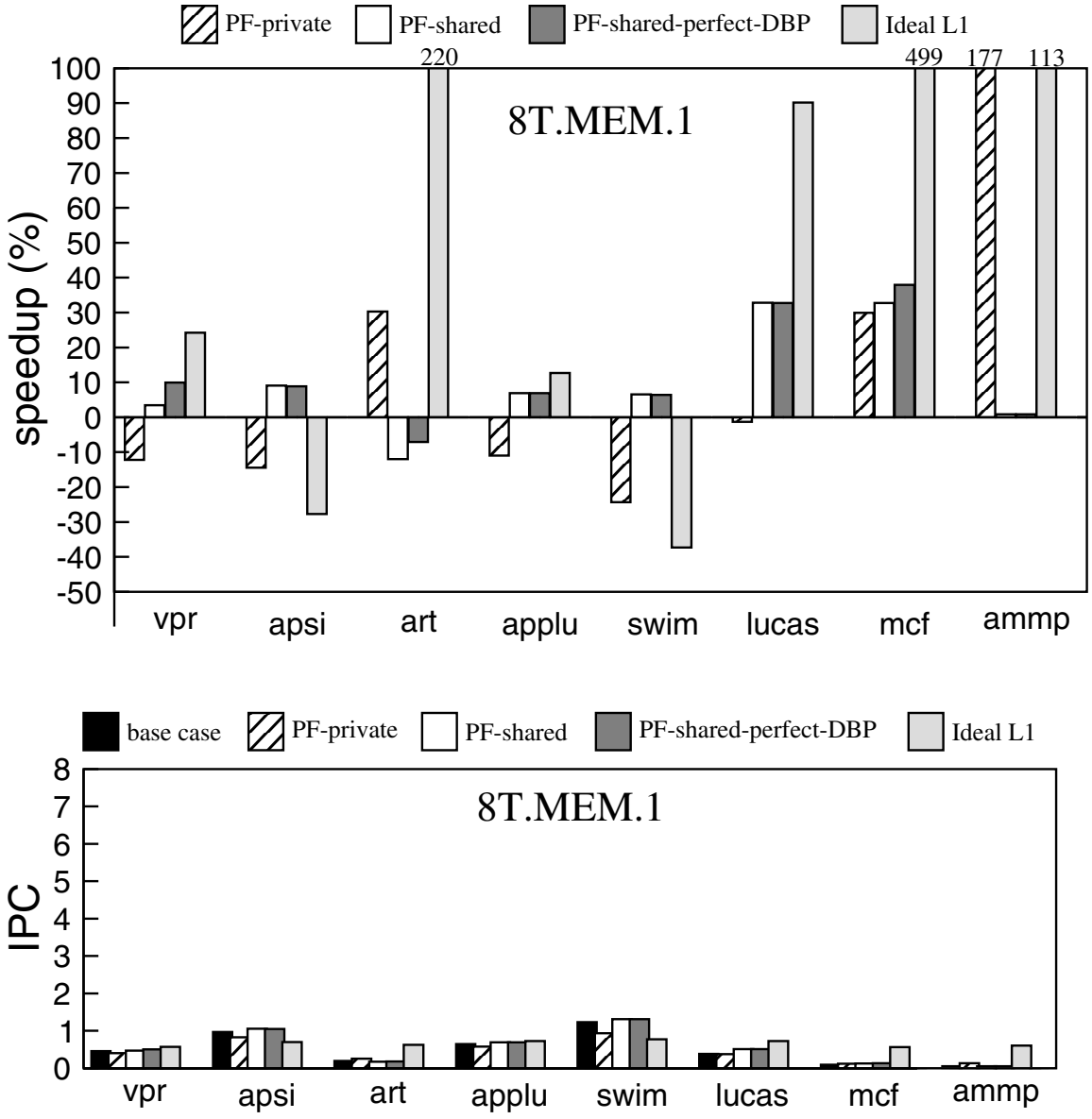


Figure 47: Detailed IPC and speedup of PF for 8T.MEM

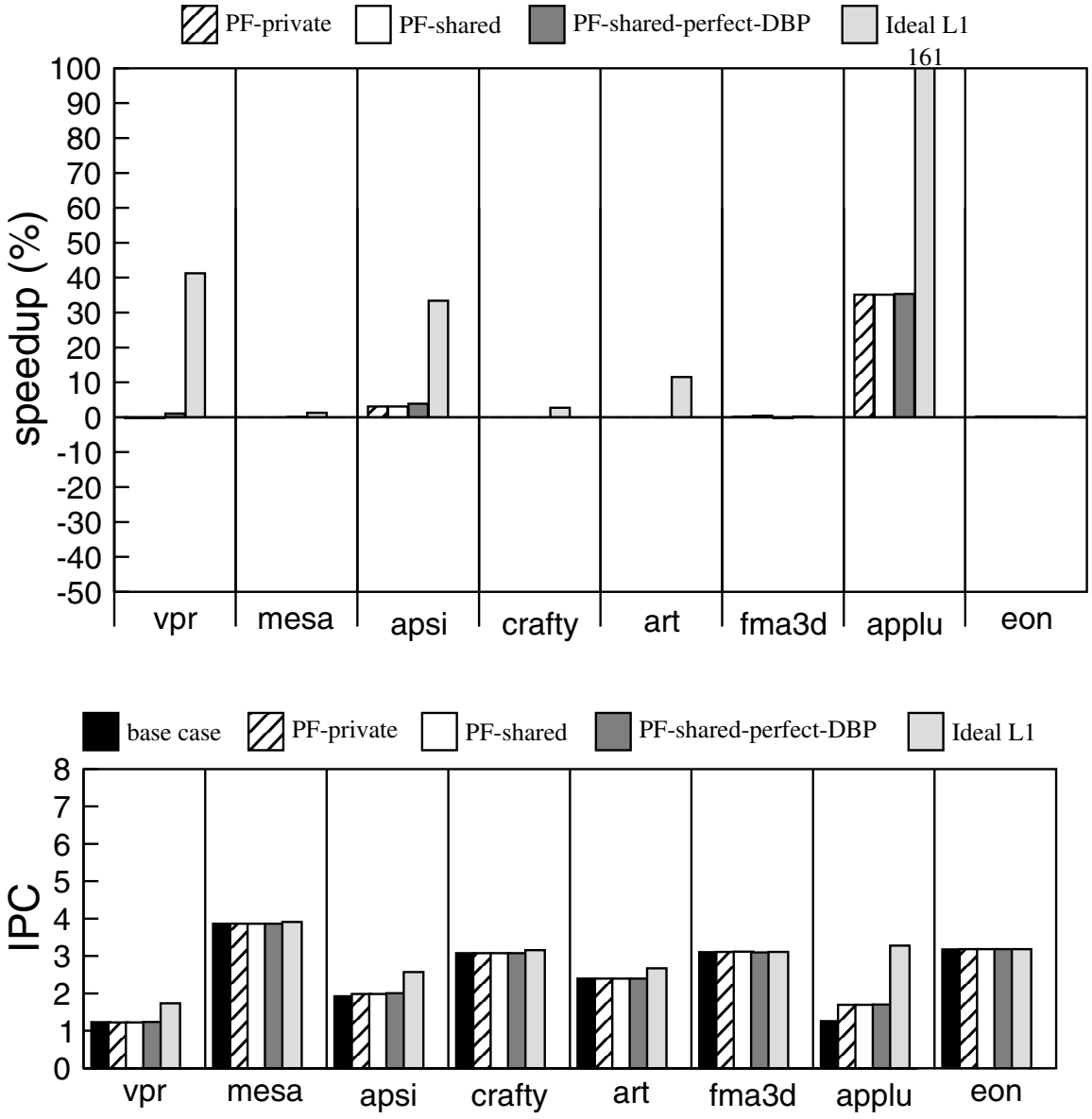


Figure 48: Detailed IPC and speedup of PF for 1T.MIX (part I)

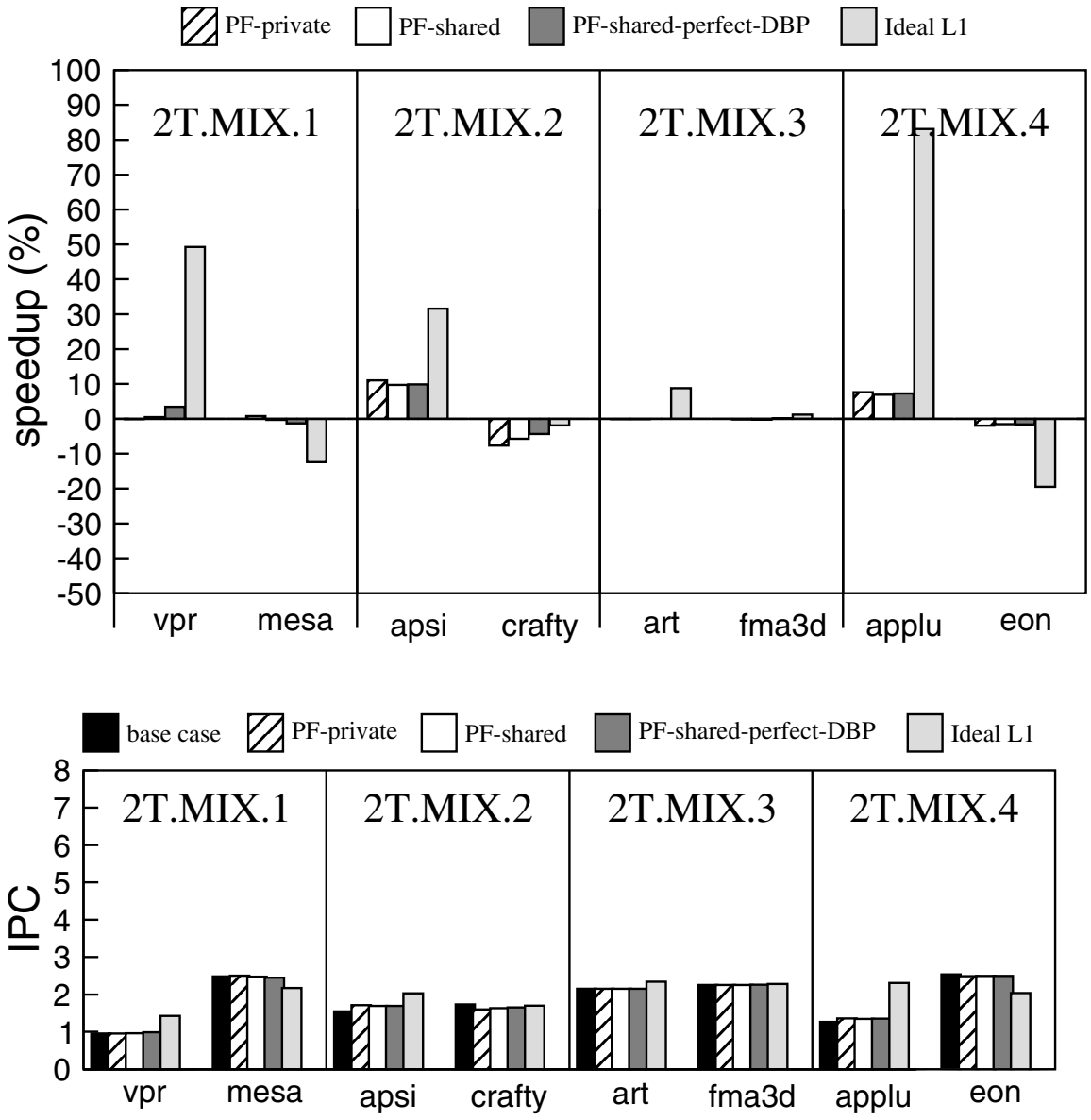


Figure 49: Detailed IPC and speedup of PF for 2T.MIX.{1,2,3,4}

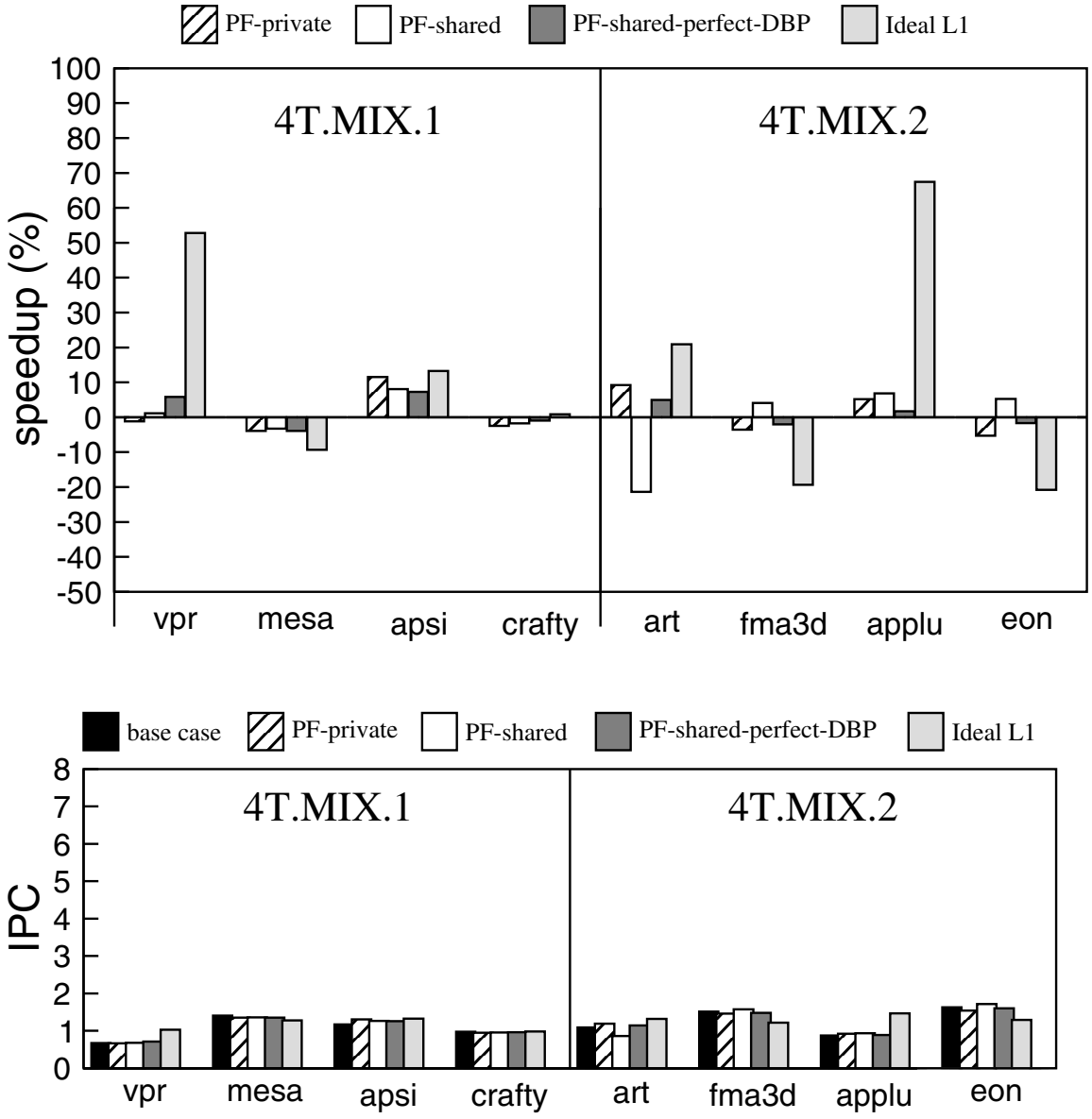


Figure 50: Detailed IPC and speedup of PF for 4T.MIX.{1,2}

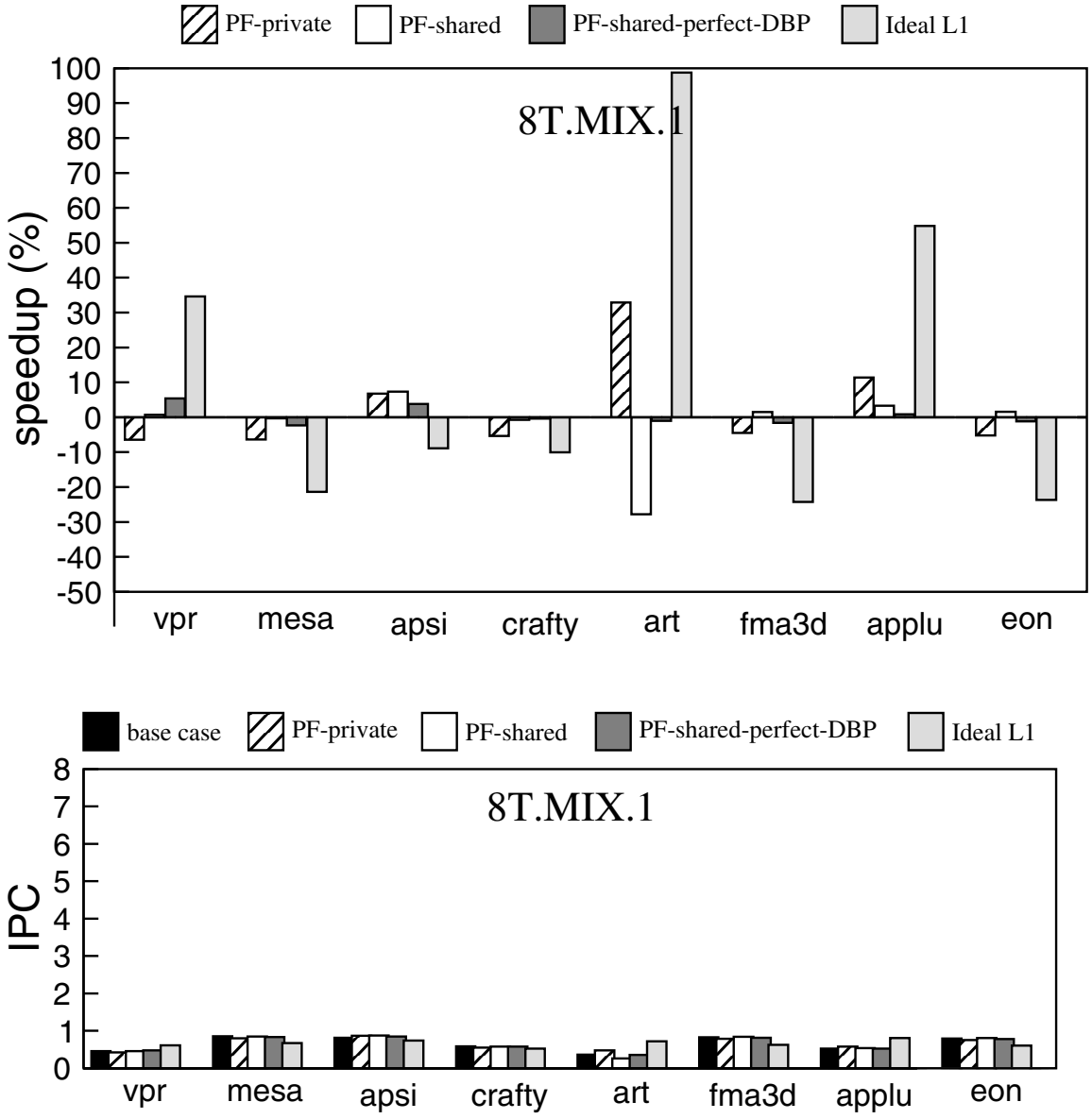


Figure 51: Detailed IPC and speedup of PF for 8T.MIX.1

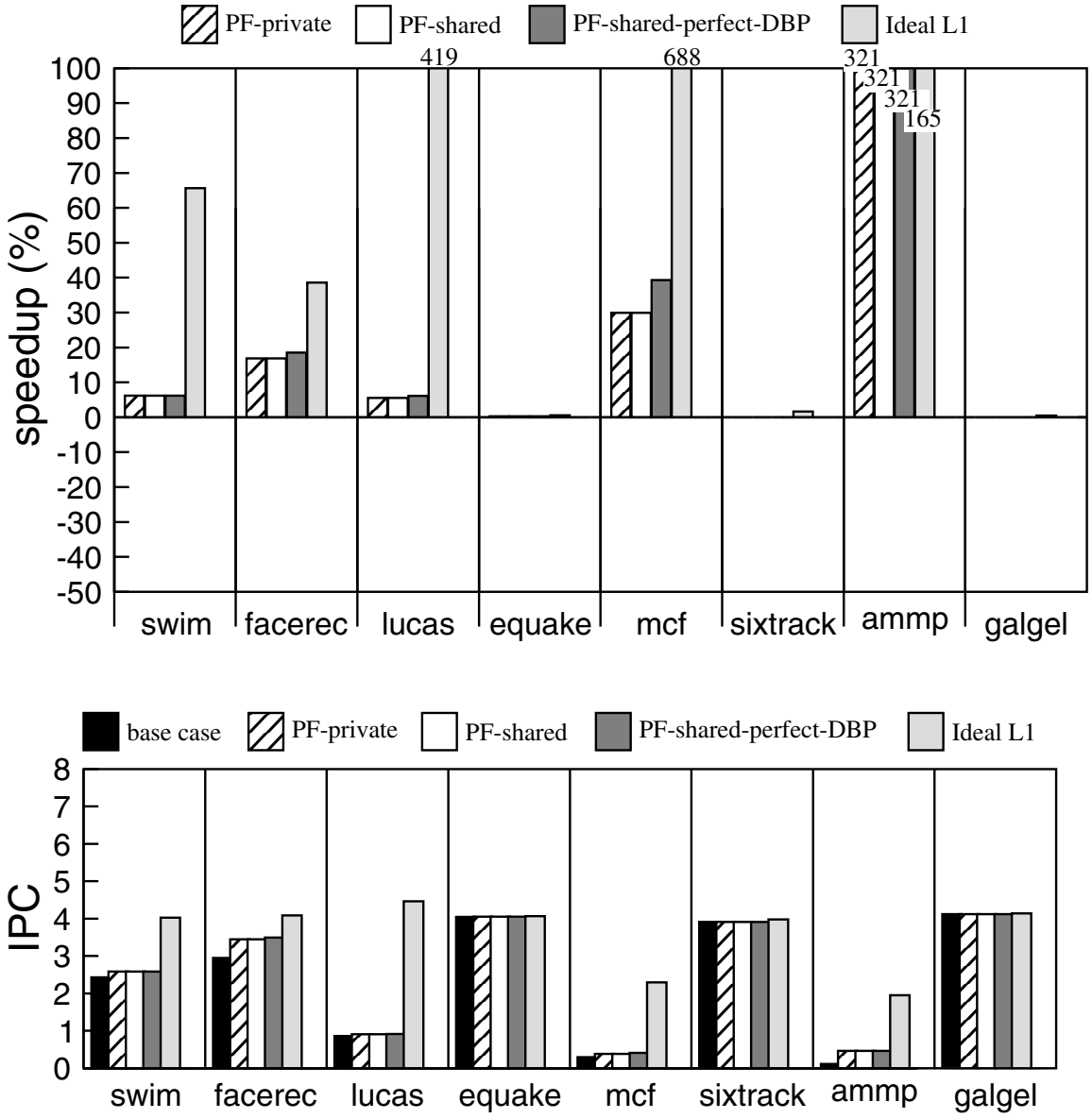


Figure 52: Detailed IPC and speedup of PF for 1T.MIX (part II)

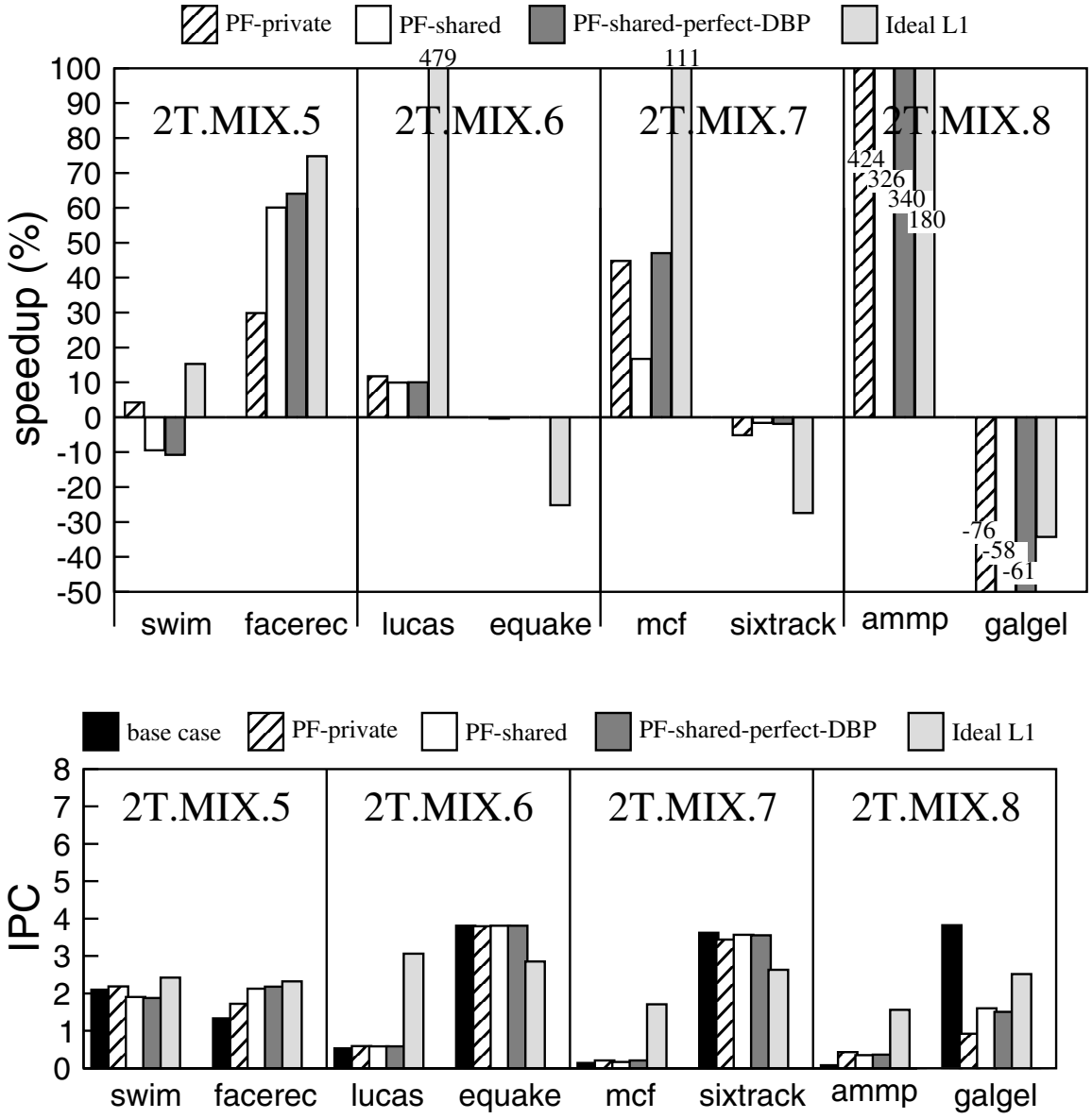


Figure 53: Detailed IPC and speedup of PF for 2T.MIX.{5,6,7,8}

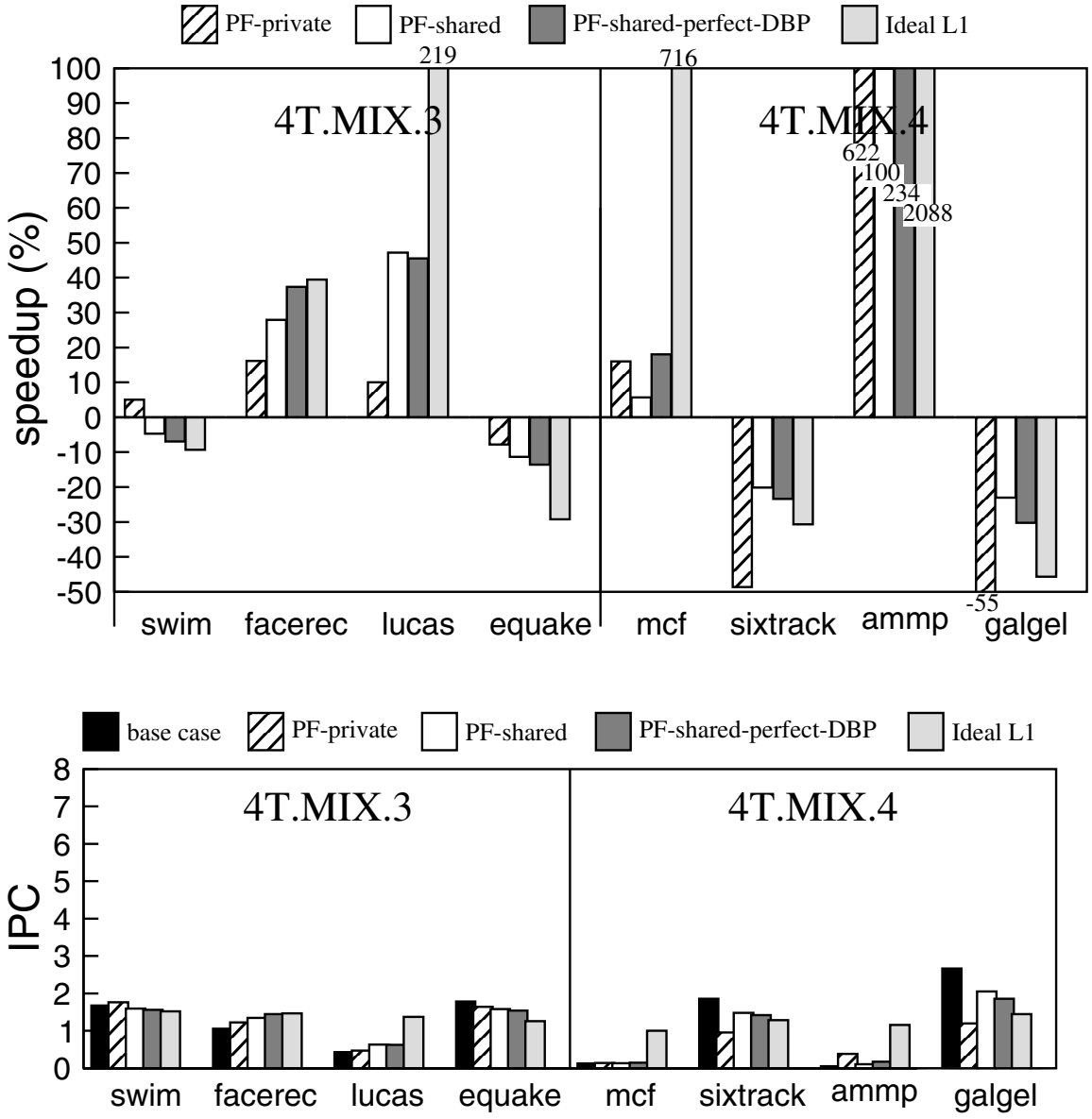


Figure 54: Detailed IPC and speedup of PF for 4T.MIX.{3,4}

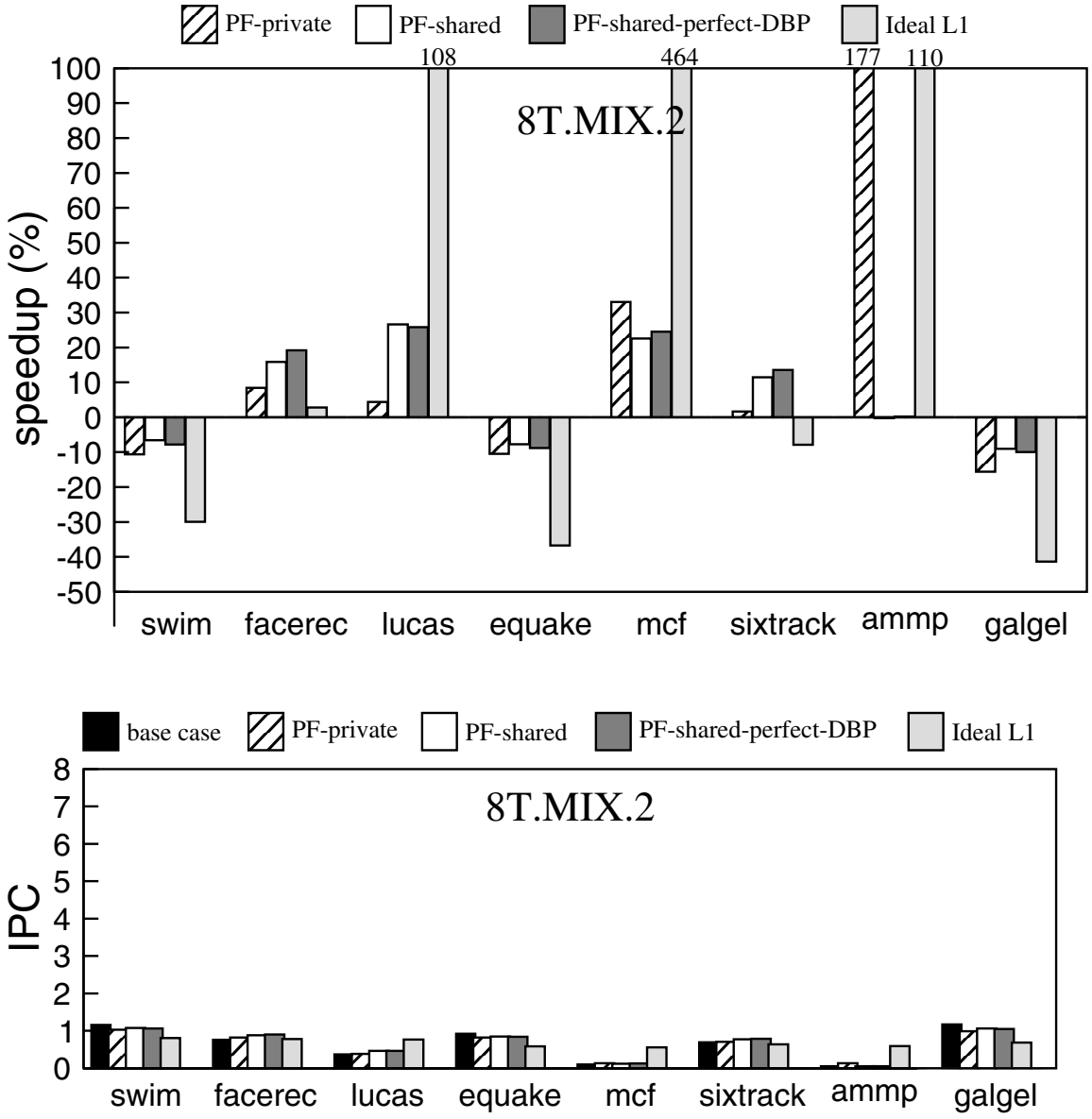


Figure 55: Detailed IPC and speedup of PF for 8T.MIX.2

5.4 Skipper Results

5.4.1 Performance of Skipper

In this section, I present the base performance of Skipper, compared to an out-of-order superscalar with branch prediction. In the two left bars, I vary the instruction window size from 128 to 256 entries. The speedups are normalized against an out-of-order superscalar with the same instruction window size. The numbers above the bars are the speedups for a 128-entry superscalar with perfect branch prediction, to serve as an idealized reference. Because Skipper uses extra storage for SCIT and JRS (about 6KB) compared to a superscalar, I also show Skipper's speedups normalized with respect to a "large superscalar" using an extra 6KB in larger prediction tables (16K entries each table, total size 12KB) in the two right bars. For both Skipper and superscalar, I use one port for the 128-entry instruction window and two ports (and double front-end width for decode and rename stages) for the 256-entry instruction window. Each cycle only one block can be fetched through one port and the entire block may not be useful due to branches and jumps within the block. For the 256-entry Skipper and superscalar, I assume aggressive front-ends that can obtain two fetch PCs from the branch predictor and use both the ports for fetching.

From the first two bars in Figure 56, I can see that for a 128-entry instruction window, Skipper achieves a wide range of speedups up to 15% for *ijpeg*, 14% for *li, 9% for go,* and 8% for *m88ksim*, all the way down to small slowdowns for *perl* and *compress*. These speedups indicate that Skipper successfully skips around difficult branches and overlaps branch resolution with control-flow independent instructions. With a 256-entry instruction window, Skipper achieves higher speedups, resulting in up to 22% speedups for *ijpeg* and *li,* 16% for m88ksim, and small speedups for *compress*. Increasing the instruction window size improves speedups because Skipper uses the extra entries better than a conventional superscalar. While Skipper brings more useful instructions into the extra entries in the instruction window, conventional superscalar is limited by mispredictions and squanders the extra entries on incorrect instructions.

The two right bars show Skipper's speedups normalized against the "large superscalar". Comparing the left bars with the right bars for the same instruction window size, we

see that the change in Skipper's speedups is less than 3% in all cases. These results indicate that the extra prediction storage does not give superscalar much performance advantage and is better used by Skipper.

5.4.2 Effectiveness of Skipper's mechanisms

Skipper's speedups widely vary across benchmarks and are still far lower than those for perfect branch prediction; the measurements in Table 9 explain the reasons. *JRS coverage* (related to the metrics in [14]) is the ratio of the number of branches JRS identifies as difficult to the total number of mispredicted branches. *Heuristic accuracy* is the ratio of the number of branches with correctly-determined reconvergence PC to the total number of branches with reconvergence PC within the gap-length-threshold. *Actual coverage* is the ratio of the number of actually skipped branches to the total number of mispredicted branches. Actual coverage measures the opportunity exploited by Skipper. JRS coverage attenuates to actual coverage due to both mispredicted branches having reconvergence PCs beyond the gap-length-threshold and the heuristic determining reconvergence PCs incorrectly.

Overshoot is the ratio of the number of skipped branches which would have been correctly predicted in a superscalar to the total number of branches. Overshoot measures unnecessary stalling. *Reconvergence accuracy* is the ratio of the number of successfully skipped branches to the number of actually skipped branches. A successfully skipped branch is one for which the reconvergence PC is reached within the instruction window gap, and there are no squashes due to in-gap branch mispredictions (Section 3.2.3), incorrect outputreg set, or skipped stores (Section 3.2.1). Reconvergence accuracy measures the accuracy of SCIT information learnt by Skipper. Skipper's misprediction rate is the ratio of the number of incorrectly predicted and unsuccessfully skipped branches to the total number of branches.

We see that overshoot is mostly less than about 11% and reconvergence accuracy is usually higher than 95%, but actual coverage is low. While JRS coverage is about 78%-98%, actual coverage falls within a mere 17-58%. I experimented with JRS's parameters

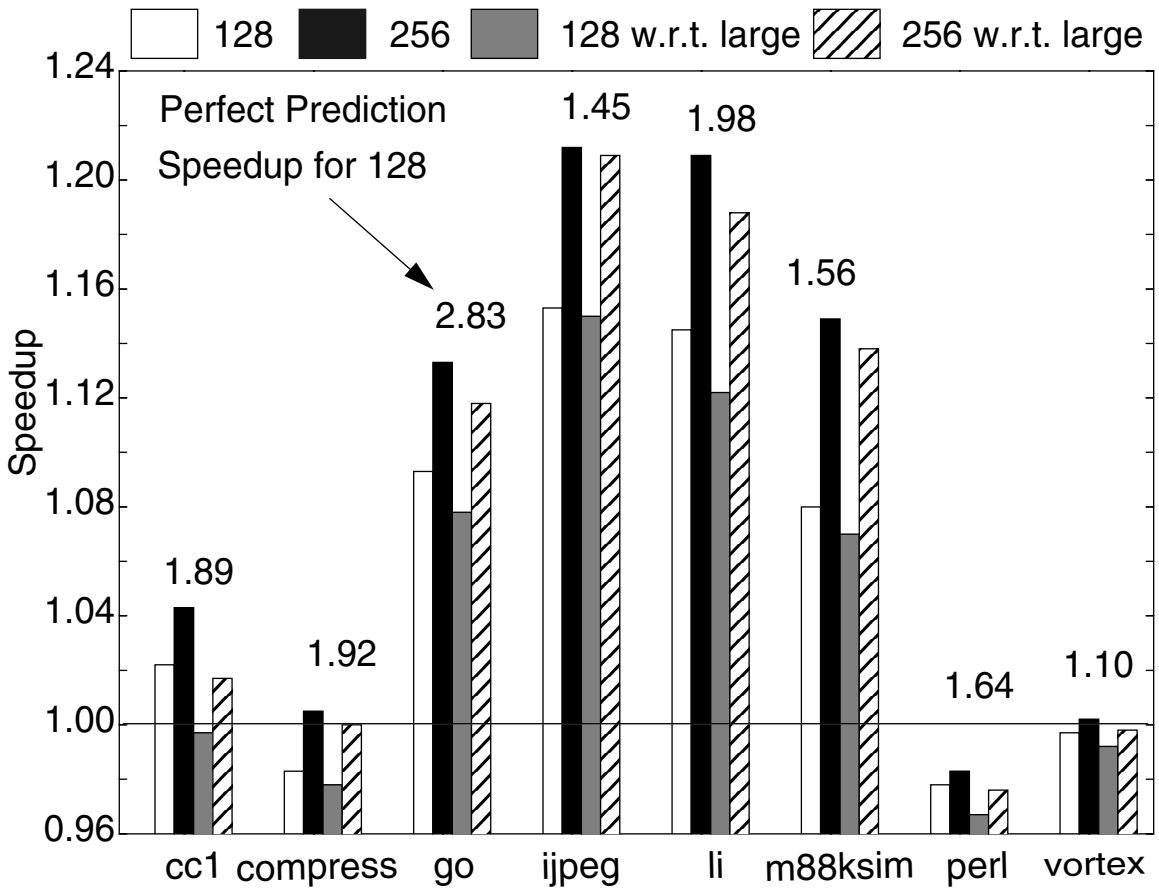


Figure 56: Base performance of Skipper using one i-cache port.

but could not obtain significantly better JRS coverage for *cc1*, *ijpeg*, *m88ksim*, and *vortex*. Actual coverage can fall far below JRS coverage due to either poor heuristic accuracy or reconvergence PCs being farther than the gap-length-threshold. Heuristic accuracy is 75%-100%, which is too high to degrade actual coverage by a large margin, implying that gap-length-threshold prevents a large fraction of difficult branches from being skipped. I vary gap-length-threshold in Section 5.4.6, but found that many difficult branches have far away reconvergence points (more than 200 instructions), requiring inordinate gap-length-threshold values. Missed opportunity due to low actual coverage is the key reason for Skipper lagging far behind perfect prediction.

cc1 and *go* incur many mispredictions both within and outside skipped computations (gaps). These mispredictions are not caught by Skipper due to its low coverage and cause squashing of post-reconvergent computation, nullifying Skipper's advantage. *compress*

Table 9. (a) Measurements of Skipper's mechanisms.

Benchmarks	JRS coverage (%)	Heuristic accuracy (%)	Actual coverage (%)	Overshoot (%)
cc1	92	75	19	7
compress	98	100	25	9
go	98	87	20	11
ijpeg	90	96	58	8
li	96	77	17	6
m88ksim	78	90	32	11
perl	94	98	16	2
vortex	88	79	17	2

Table 9. (b) Measurements of Skipper's mechanisms (continue)

Benchmarks	Reconvergence accuracy (%)	Skipper's misprediction rate (%)	Superscalar misprediction rate (%)
cc1	92	8	10
compress	100	8	12
go	89	16	24
ijpeg	98	3	9
li	100	4	8
m88ksim	99	2	4
perl	100	3	4
vortex	98	1	1

Table 10. Gap characteristics.

Benchmark	#gaps	outputreg	#instrs	#slots
cc1	1.4	4	7	14
compress95	1.5	3	4	10
go	1.4	5	10	21
ijpeg	2.0	4	5	13
li	1.2	3	9	16
m88ksim	2.1	2	5	9
perl	1.3	3	4	8
vortex	1.0	3	8	13

runs out of instruction window slots for a 128-entry instruction window, alleviated only slightly by a 256-entry instruction window. In addition to mispredictions, *go* also incurs many memory dependence squashes (*go* is the only benchmark with this problem). *ijpeg* and *m88ksim* have higher coverages than the rest, translating to higher performance. In *li*, Skipper skips *entire* short unpredictable loops (dynamic instructions in all iterations less than 10). Because loop back branches within skipped loops in *li* are not predicted but suspended till resolution, *li* avoids many mispredictions, achieving high speedups. *li*'s coverage is small because the coverage numbers do not include such suspended branches which are not mispredicted but not skipped either. *Perl* has many non-return, indirect jump mispredictions both within and outside skipped computations, which have the same effect as *cc1*'s mispredictions. *Vortex*'s prediction accuracy is high, leaving little opportunity for Skipper.

5.4.3 Characteristics of skipped computations

Table 10 shows the average number of actually skipped branches in flight ("#gap" column), inputreg and outputreg registers per difficult branch ("#in" and "#out" columns), dynamic instructions per skipped computation ("#instr" column), and the average instruction window gap length, not including pmoves ("#slot" column). The benchmarks have less than two difficult branches in flight, implying that only two skipped computations

need to be tracked in the GILB and SIST. The number of skipped instructions ranges between four and ten, even though the gap-length-threshold is 48, implying that the threshold is not hit often. The difference between the gap lengths and the number of skipped instructions is about seven, implying that Skipper wastes only a few instruction window slots. The number of outputreg registers being about four means that Skipper inserts around four extra pmoves, which could execute together in one cycle on a 4-way issue machine. The number of inputreg and outputreg registers together is about ten implying that Skipper needs to handle only ten rename maps per skipped branch. In comparison, rename tables in a 4-way issue machine handle 12 registers (8 sources and 4 destinations) every cycle, suggesting that Skipper incurs low rename bandwidth overhead.

5.4.4 Comparison between Skipper and Polypath

In this section, I compare Skipper against the previously-proposed Polypath architecture [22,23]. In Figure 57, I vary the configuration from 128 instruction window entries with one i-cache port (two left bars) to 256 entries with two i-cache ports (two right bars). The speedups are all normalized to a superscalar with equal instruction window size and equal number of i-cache ports.

For Polypath, I use the fetch policy and JRS parameters recommended in [22,23]. My model of Polypath is different than those in [22,23] in two ways, which affect its speedups. First, the Polypath papers compare a Polypath system using two i-cache ports and double pipeline width for decode and rename stages, with a superscalar using one i-cache port. However, such a comparison fails to isolate the impact of the architecture from the impact of the fetch bandwidth. Therefore, I assume exactly equal fetch bandwidth for Skipper, Polypath, and superscalar. Second, the Polypath papers do not charge any extra cycles to copy the entire rename table (the equivalent of inputreg set) needed to execute both paths of difficult branches. Because of the arguments given in [24] and in Section 3.1.1, I charge cycles for this copying as per the bandwidth of the rename tables. This charging is done for both Skipper and Polypath.

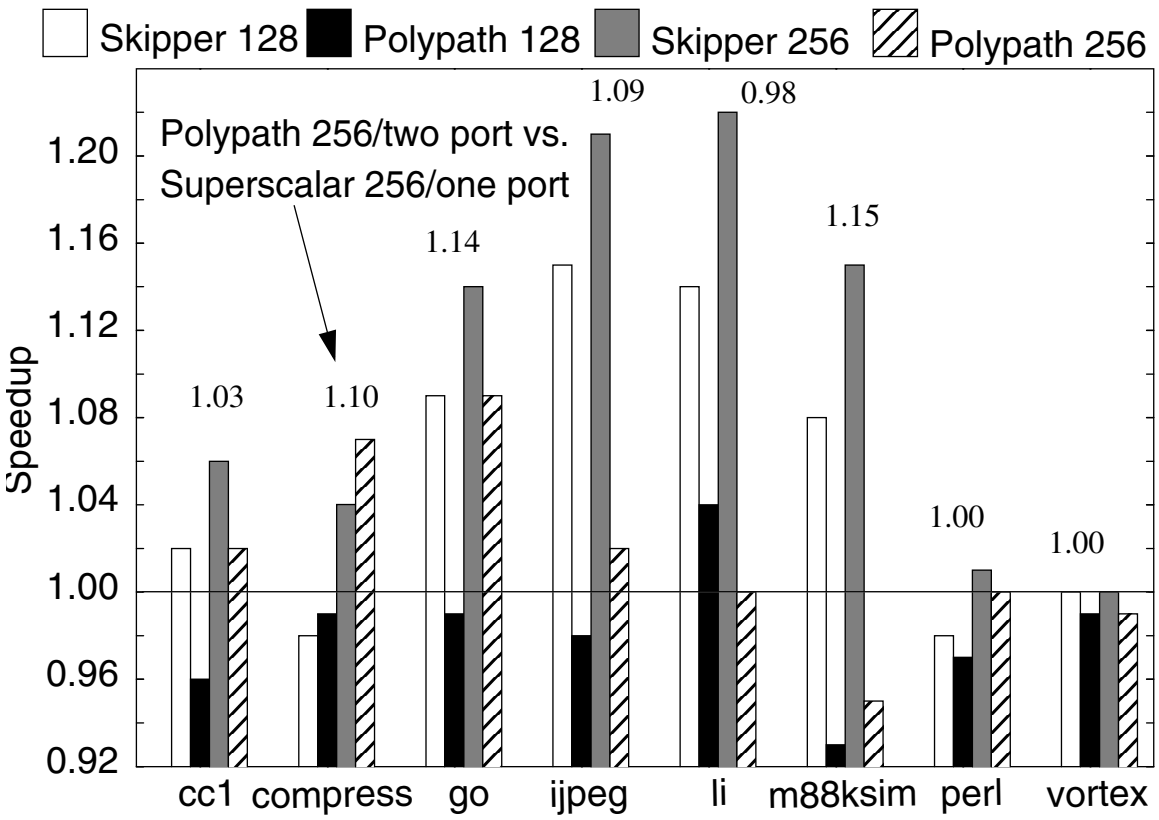


Figure 57: Comparison between Skipper and Multipath.

As before, for the 256-entry case, I assume an aggressive superscalar that can obtain two fetch PCs from the predictor and use both the i-cache ports. If there are no difficult branches in flight, both Skipper and Polypath use the ports exactly the way superscalar does. Polypath fetches both paths of difficult branches. Skipper fetches the post-reconvergent stream and the control-flow dependent instructions if there are any resolved skipped branches, and defaults to superscalar mode, if none of the skipped branches are resolved. I also compare a two-port Polypath without charging cycles for rename table copying, against a one port superscalar and show the numbers above the bars.

From the left two bars in Figure 57, we see that for the 128-entry case, Skipper outperforms Polypath significantly for *ijpeg*, *m88ksim*, *li*, and go, and modestly or not at all for the other benchmarks. Polypath achieves no speedups mainly because with only one i-cache port, there is not enough bandwidth to fetch down both taken and not-taken paths on difficult branches. This experiment clearly shows that Skipper achieves speedups because

Table 11. base case's IPC.

Benchmark	**base 128**	**base 256**	**base 256 with one-port**
cc1	1.85	1.87	1.85
compress95	2.40	2.46	2.42
go	1.51	1.55	1.51
ijpeg	3.58	3.82	3.60
li	2.10	2.14	2.14
m88ksim	2.82	3.42	2.82
perl	2.39	2.56	2.57
vortex	3.00	3.18	3.16

of much more efficient use of i-cache bandwidth than Polypath. From the two right bars, we see that for the 256-entry, two i-cache port case, Skipper outperforms Polypath significantly for go, *ijpeg*, *m88ksim*, and *li*, similar to the 128-entry case. In the case of *compress*, Polypath performs better than Skipper by 3%. Further investigation reveals that *compress* has dense data dependencies, disallowing any overlap of post-reconvergent instructions. Because Polypath executes the skipped instructions without any delay unlike Skipper, compress benefits from Polypath.

Compared to the 128-entry, one-port case, Skipper achieves even higher speedups using 256 entries and two ports, with the exception of *vortex*, indicating that Skipper can better use higher i-cache bandwidths than a superscalar. Also, a two-port Polypath with no rename table copy overhead achieves speedups compared to a one-port superscalar, as shown in previous papers [22,23].

Table 11 lists the instructions per cycle (IPC) for the base cases for Figure 57 (also used for Figure 56 and later graphs). The first column lists the IPC for the base cases with 128-entry instruction window with one i-cache port, which the base case is used for the first and second bars in Figure 57. Similarly, the second column lists the IPC for the base

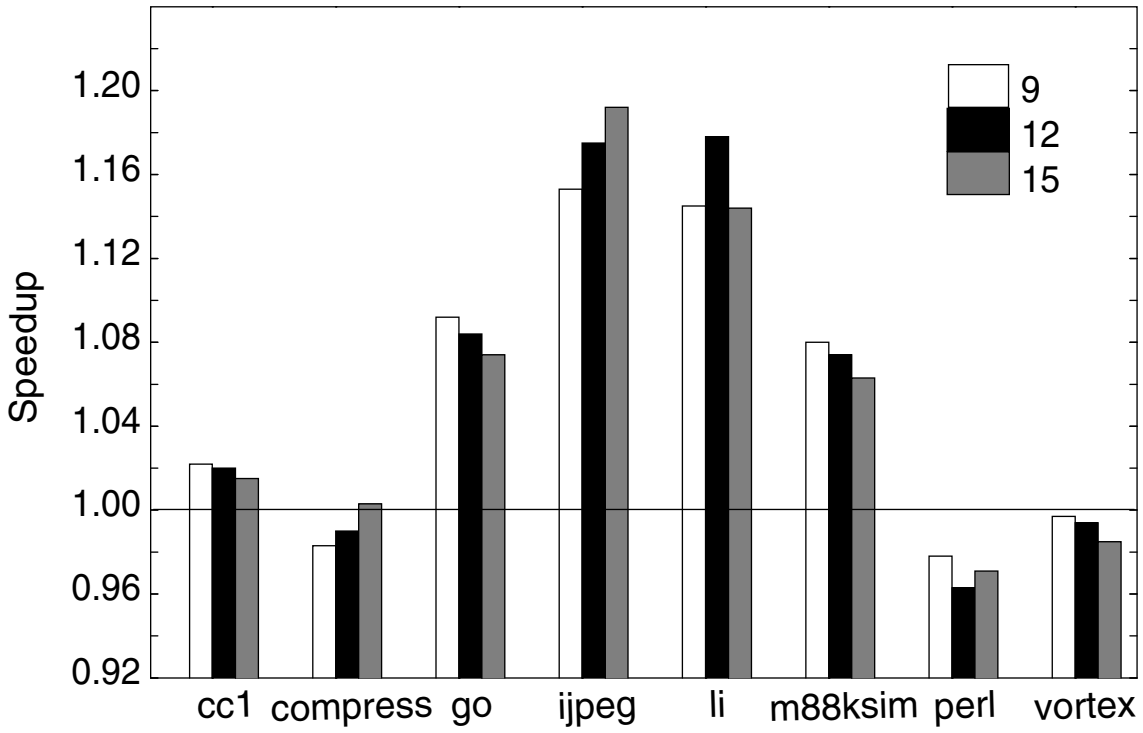


Figure 58: Effect of Skipper's misprediction penalty.

cases with 256-entry instruction window with two i-cache ports, which the base case is used for the third and fourth bars. The third column lists the IPC of base cases with 256-entry instruction window with only one i-cache port, which the base cases is used for the speedups shown in text above each benchmark in Figure 57. The IPC for these benchmarks are fairly high for a eight-issue superscalar, showing that my methodology does not artificially suppress the base cases' performance to allow more potential improvement. Comparing second and third column, I also notice that base cases with two i-cache ports have much higher IPCs than one. This shows that the base cases can take advantage of the second i-cache ports, reducing Polypath's opportunity and thus explaining why the speedup for the Polypath is low in my evaluation (fourth bar) than in [23] (text above bars). For 256-entry instruction window, *Perl's* IPC for slightly lower for base cases with two ports than with one, because of slight higher branch misprediction ratio in the former case.

5.4.5 Misprediction Penalty

To see the effect of deepening pipelines, I varied misprediction penalty as 6, 9, and 12 cycles in Figure 58. On one hand, a longer misprediction penalty gives Skipper the opportunity to achieve higher speedups by eliminating the more-expensive mispredictions. On the other hand, a longer misprediction penalty forces Skipper to find more data independent, post-reconvergent instructions to execute before the difficult branch can fill the pipeline with the correct control-flow dependent instructions. Thus, it is a conflict between opportunity and data independence. We see two trends in speedups on increasing penalty: One in which Skipper's speedups for *ijpeg*, *li*, and *compress* indicating that opportunity overcomes dependencies in these benchmarks. And the other in which Skipper's speedups for the rest of the benchmarks reduce due to dependencies offsetting opportunity. Skipper's low coverage restricts opportunity to avoid mispredictions.

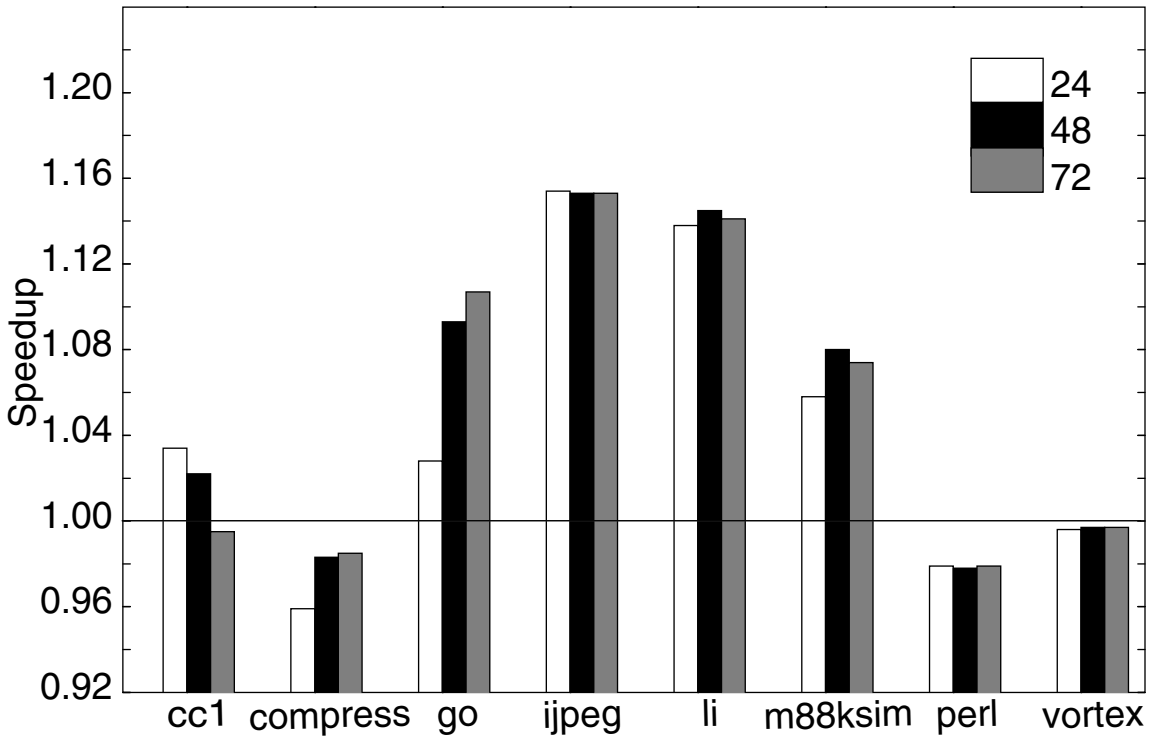


Figure 59: Effect of Skipper's gap-length-threshold.

5.4.6 Effect of gap-length-threshold

Because the analysis in Section 5.4.3 indicates that gap-length-threshold impacts coverage, I varied the gap-length-threshold as 24, 48, and 72 in Figure 59. While a larger threshold allows better actual coverage, larger threshold also allows branches with larger gap lengths to be skipped, incurring wasted instruction window and load/store queue slots. Thus, it is a trade-off between coverage and instruction window utilization. Increasing the threshold from 24 to 48 improves coverage for all the benchmarks by about 1%-6% reaching the values shown in Table 9. I found that except for *go*, the rest of the benchmarks are not affected by increasing the gap-length-threshold beyond 48. Increasing the threshold from 48 to 72, go's speedup improves from 8% to about 11%. This experiment shows that Skipper's actual coverage is limited not by the threshold setting but by long gaps inherent in programs. I also varied the SCIT size as 32, 128, and 512 entries and found that increasing the SCIT size beyond 128 entries does not improve speedups. Because the base prediction accuracy is high, only a few (static) branches are identified as difficult and they fit within 128 entries.

6 CONCLUSIONS

In this dissertation, I evaluated trace cache, value prediction and prefetching in SMT. I found that SMT's sharing of the instruction storage (i.e., trace cache or i-cache), physical registers, and issue queue impacts the effectiveness of trace cache, value prediction, and prefetching, respectively. Also, in order to reduce branch misprediction penalty on superscalar, I propose a novel architecture, called Skipper, to handle difficult branches. Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in the context of a superscalar pipeline.

I found that: (1) Trace cache introduces multiple copies of the same instructions in different traces, trading off space for bandwidth. However, SMT needs a large instruction storage because multiple threads share the storage. Furthermore, trace cache's benefit of supplying many instructions in one fetch diminishes in SMT because SMT can do the same with conventional instruction cache by fetching from multiple threads. My simulations showed that when compared to a similar-sized i-cache, trace cache's space-for-bandwidth trade-off degrades SMT throughput. For two threads, trace cache improves throughput. thus supporting Intel's decision to use a trace cache in the two-threaded Pentium IV. (2) Value prediction causes hold-up of physical registers and cannot release them until after the predicted instruction completes and commits. Because SMT's multiple threads share physical registers, this hold-up stalls progress in other threads. Thus, unlike superscalar, SMT incurs throughput degradation even with correct value predictions. My simulations showed that with a typical number of physical registers, value prediction degrades SMT throughput; and with unlimited registers, value prediction's benefit disappears with an increasing number of threads. (3) Prefetching into L2 converts slow L2 misses into fast L2 hits. However, the L2 hits still miss in L1, resulting in the same L1 misses occurring in fewer cycles. Because instructions dependent on the L1 misses clog

the issue queue and because SMT's multiple threads share the issue queue, this clogging stalls progress in other threads. With prefetching, L1 misses occur in fewer cycles, clogging the issue queue more often. Thus, unlike superscalar, SMT incurs throughput degradation even with correct prefetches. Therefore, SMT needs to balance prefetching and issue queue clogging. My simulations showed that prefetch coverage can be reduced to achieve such balance, improving throughput for memory-intensive workloads. However, for workloads with mixed memory demand (high-ILP and memory-intensive threads), prefetching has little opportunity and slightly degrades throughput.

Because trace cache and value prediction hurt SMT throughput, I recommend that these techniques to be excluded for future SMT designs when multi-programmed workloads are the common case and throughput is the main goal. For SMT designs where both single-threaded and multi-programmed workloads are common, my results introduce a challenging design dilemma for SMT designers: on one hand, the techniques are ineffective for multi-programmed workloads and in many cases hurt throughput; on the other hand, the techniques significantly improve single-thread performance, and disabling them to improve multi-programmed throughput would hurt single-thread performance.

My findings also create a new responsibility for the OS: Because these techniques improve single-thread performance, I recommend that the OS disable the techniques when running multi-programmed workload and enable them for single-threaded workload. In the environment where threads have different priorities, because the techniques may improve low priority threads while hurting higher priority threads, I recommend that the OS should selectively enable the techniques only for high priority threads.

Skipper exploits control-flow independence by skipping over control-flow dependent computation of frequently mispredicted branches, in the context of a superscalar pipeline. Skipper fetches the skipped control-flow dependent instructions after the post-reconvergent instructions, out of program order. I describe key mechanisms to implement Skipper without unduly complicating the pipeline despite out-of-order fetch, including (1) identifying difficult branches using the previously-proposed JRS scheme, (2) determining the difficult branch's reconvergence point without scanning, (3) handling out-of-order

fetching of the skipped instructions but maintaining program order in the instruction window, and (4) handling data dependencies among the skipped instructions and the yet-to-be fetched post-reconvergent instructions using the existing register rename tables and load/store queue. SPECint95 simulations show that Skipper performs 10% and 8% better than superscalar and the previously-proposed Polypath, respectively, when all three microarchitectures use a 256-entry instruction window and two i-cache ports.

LIST OF REFERENCES

LIST OF REFERENCES

[1] H. Akkary and M. A. Driscol. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, Nov. 1998.

[2] A.Mendelson and F.Gabbay. Speculative execution based on value prediction. Technical report, Technion, 1997.

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei Hwu. Integrated predicated and speculative execution in the impact epic architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 237–248, Nov. 2001.

[5] B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Oct. 1999.

[6] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the Eighth International Symposium on High Performance Computer Architecture (HPCA)*, pages 83–94, Feb. 2002.

[7] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 27)*, pages 181–190, Nov. 1994.

[8] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors-the simplescalar tool set. In *Technical Report 1308, University of Wisconsin Madison Computer Sciences Department*, July 1996.

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

[10] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.

[11] Y. Chou, J. Fung, and J. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 1999 International Conference on Supercomputing*, June 1999.

[12] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Apr. 1994.

[13] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, Nov. 1997.

[14] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–131, June 1998.

[15] E. Hao, P.-Y. Chang, and Y. Patt. The effect of speculatively updating branch history on branch prediction accuracy. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 27)*, pages 228–232, Nov. 1994.

[16] Z. Hu, M. Martonosi, and S. Kaxiras. Tcp: Tag correlating prefetchers. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2003.

[17] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, pages 312–320, Dec. 1996.

[18] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, June 1997.

[19] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[20] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behaviour to reduce cache leakage power. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, June 2001.

[21] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the Sixth International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[22] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, pages 38–47, Nov. 1999.

[23] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–259, June 1998.

[24] M. J. Knieser and C. A. Papachristou. Y-pipe: A conditional branching scheme without pipeline delays. In *Proceedings of the 25th annual international symposium on Microarchitecture (MICRO)*, pages 125–128, Dec. 1992.

[25] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, June 2001.

[26] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid:software prefetching in pointer and call intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 28)*, Nov. 1995.

[27] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[28] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture (MICRO)*, pages 45–54, Dec. 1992.

[29] S. McFarling. Combining branch predictors. In *Technical Report TR-36, DEC-WRL*, June 1993.

[30] M.H.Lipasti, C.B.Wilkerson, and J.P.Shen. Value locality and data speculation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 138–147, Oct. 1996.

[31] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, pages 181–193, June 1997.

[32] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, Dec. 1997.

[33] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 171–182, Nov. 2002.

[34] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

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

[36] S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechanism. In *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*, 1998.

[37] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, Dec. 1996.

[38] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, 1999.

[39] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, Nov. 1999.

[40] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin-Madison, Dec. 1997.

[41] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, pages 194–205, June 1997.

[42] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA)*, pages 414–425, June 1995.

[43] J. Stark, P. Racunas, and Y. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation statios out-of-order. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 257–266, Dec. 1997.

[44] T.C.Mowry, M.S.Lam, and A.Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Oct. 1992.

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

[46] G. H. Timothy Sherwood, Erez Perelman and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[47] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, Dec. 2001.

[48] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[49] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA)*, pages 392–403, June 1995.

[50] G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 218–227, Dec. 1997.

[51] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 81–92, June 1998.

[52] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[53] T. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th annual international symposium on Computer architecture (ISCA)*, May 1993.

[54] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing instruction fetch rate via multiple branch prediction and a branch address cache,. In *In Proc. of the 7th ACM Int. Conf. on Supercomputing*, pages 67–76, July 1993.

[55] S. K. Zhigang Hu and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA)*, May 2002.

VITA

VITA

Education:
High School (1988-1994)
Chong Ling High School, Ayer Item, Penang MALAYSIA

BS (1995-1998) Purdue University, West Lafayette, IN USA
Major: Computer Engineering

MS (1998-2000) Purdue University, West Lafayette, IN USA
Major: Electrical and Computer Engineering,
Primary Area: Computer Architecture

PhD (2000-May 2004) Purdue University, West Lafayette, IN USA
Major: Electrical and Computer Engineering
Primary Area: Computer Architecture

Publication:
Hai Li, Chen-Yong Cher, T. N. Vijaykumar, and Kaushik Roy,
VSV: L2-miss-driven variable supply-voltage scaling for low power.
In Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO), December 2003.

Chen-Yong Cher and T. N. Vijaykumar,
Skipper: A microarchitecture for exploiting control-flow independence.
In Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO), December 2001.