TRACE PROCESSORS:

EXPLOITING HIERARCHY AND SPECULATION

by

Eric Rotenberg

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1999

Abstract

In high-performance processors, increasing the number of instructions fetched and executed in parallel is becoming increasingly complex, and the peak bandwidth is often underutilized due to control and data dependences. A *trace processor* 1) efficiently sequences through programs in large units, called *traces*, and allocates trace-sized units of work to distributed processing elements (PEs), and 2) uses aggressive speculation to partially alleviate the effects of control and data dependences. A trace is a dynamic sequence of instructions, typically 16 to 32 instructions in length, which embeds any number of taken or not-taken branch instructions. The hierarchical, trace-based approach to increasing parallelism overcomes basic inefficiencies of managing fetch and execution resources on an individual instruction basis.

This thesis shows the trace processor is a good microarchitecture for implementing wide-issue machines. Three key points support this conclusion.

1. *Trace processors perform better than wide-issue superscalar counterparts* because they deliver high instruction throughput without significantly increasing cycle time. The underlying reason: trace processor cycle time is more sensitive to individual PE complexity than full processor complexity.

2. *The trace processor organization naturally supports aggressive speculation*. The contiguous instruction window enables high-performance, but relatively transparent, selective recovery from data misspeculation. Control flow hierarchy and existing data

speculation support are leveraged to selectively preserve traces after a mispredicted branch that are control independent of the branch.

3. *Trace processors are an evolutionary extension of superscalar processors* and, as a result, are quite viable. They retain binary compatibility and a conventional, single flow of control. Wide instruction fetching (using a single program counter) enables instructions to be scheduled quickly. The approach does not rely on sophisticated and potentially less-robust multithreading or explicitly-parallel compilers to accurately schedule instruction fetching from multiple, disjoint points in the program.

This thesis makes two main contributions. First, a trace processor microarchitecture is fully developed, including: the trace cache based, hierarchical sequencing mechanism; the distributed instruction window and hierarchical issue mechanisms; data speculation mechanisms including selective recovery support; and hierarchical control independence techniques. Second, performance evaluations and complexity analysis support the three key points listed above.

Acknowledgments

First and foremost, I would like to thank my parents, Michael and Myriam, and sisters, Dorith and Katie, for their constant support even when the demands of graduate school kept us apart.

The influence of my advisor Jim Smith is immeasurable. He has taught me so many things about computer architecture, research, and writing. Also, his quiet confidence, gems of wisdom, and wonderful view of life have transformed a part of me for the better.

My close friends have provided a much needed escape from Room 3652. Mike Andorfer has been a good friend since high school, as have Michael Tierney and Mike Bechtol since undergraduate school. During my years in graduate school, I was fortunate to cross paths with Jim Smith, Emmanuel Ackaouy, Jeremiah Williamson, Paul Thayer, Quinn Jacobson, Ravi Rajwar, Richard Carl, Jose Gonzales, Bill Donaldson, and Todd Bezenek. I cannot fully express how much their companionship, generosity, unique personalities, and antics mean to me. For their sake and my indulgence, I feel compelled to reminisce and acknowledge happy times: the Union Terrace, the Plaza, the Crystal Corner, the Inferno, the "Tremendous Twelve", serious beer drinking, grilling out, good tunes and bad movies, jamming guitar, UW hockey and women's volleyball, GBP football, tragic and comic Vitense golfing, the voice menu, and Room 3652 hijinks.

Quinn Jacobson, Yiannakis Sazeides, Timothy Heil, S. Sastry, and Craig Zilles were always willing to listen to my ideas and give excellent feedback. Our frequent day-long discussions — technical, political, and philosophical — are the essence of graduate school. I thank Quinn, Yanos, and Tim for their contributions to the trace processor simulator. I have learned much about computer architecture through my close collaboration with Quinn, whose keen mind never ceases to amaze me.

I am fortunate to have worked with many Masters and PhD students associated with the Wisconsin Multiscalar group, although I refrain from naming everyone here. Special thanks go to Todd Austin, Steve Bennett, Andy Glew, Sridhar Gopal, Erik Jacobsen, Harit Modi, Andreas Moshovos, Subbarao Palacharla, Amir Roth, and Avinash Sodani for many diverse and stimulating technical discussions. Scott Breach and T. N. Vijaykumar provided me with invaluable insight into the multiscalar paradigm. I enjoyed working closely with Matt Kupperman and Craig Zilles on the Kestrel ARB design. Outside the Multiscalar group, Ravi Rajwar maintained my link to the other half of computer architecture, multiprocessing.

The CS and ECE faculty are the reason Wisconsin is such a stimulating place to learn about computer design. Jim Goodman, Mark Hill, Chuck Kime, Parmesh Ramanathan, Kewal Saluja, Jim Smith, Guri Sohi, and David Wood have all been inspirational in my development. And I am indebted to those who served on my preliminary and final defense committees. Mark Hill and Guri Sohi put in a lot of time to improve the quality of this thesis, and throughout my studies, Guri gave me important lessons in microarchitecture.

IBM's AS/400 division in Rochester, Minnesota funded my studies for three years; special thanks go to John Borkenhagen, Mike Gruver, and Hal Kossman for supporting my research and exchanging ideas. My research was also supported in part by NSF Grant

MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346.

I would like to thank Venkatesh Iyengar for his help in the carefully orchestrated, complicated process of depositing a dissertation. Todd Bezenek provided assistance in getting runs going in the x86 Condor pool, in addition to providing competent advice on just about everything. Finally, there are the unsung heroes — the system administrators and Condor staff. The CSL provides the highest quality computing services around, and any problems I encountered were promptly and efficiently dealt with. Last, but not least, is Bruce Orchard, whose reply always seemed to arrive before I sent out the complaint, even well after midnight.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the primary reasons for the continued success of general purpose computers is the rapid, unyielding growth in microprocessor performance. Successive generations of high performance microprocessors not only reduce the run time of existing applications, but also enable new levels of functionality and software complexity.

Improvements in microprocessor performance come about in two ways - advances in semiconductor technology and advances in processor microarchitecture. It is almost certain that clock frequencies will continue to increase. The microarchitectural challenge is to extract parallelism from ordinary, sequential programs -- programs that are not written in an explicitly parallel fashion, but which nevertheless contain significant amounts of inherent parallelism at the level of individual program instructions. By exploiting *instruction-level parallelism* (ILP), processors can increase the amount of work performed in parallel and speed execution of the most prevalent class of applications.

Because there are dependences among instructions, finding a sufficient number of independent operations to execute in parallel requires examining and scheduling a large group of instructions, called the *instruction window*. The larger this window, the farther a processor may "look ahead" into the program for independent instructions.

This basic approach underlies all high performance processing models, whether compiler-oriented or hardware-intensive. The predominant high performance processing paradigm, superscalar processing, faces significant challenges as higher levels of ILP are needed. The first challenge is *microarchitectural complexity*. The circuits used to construct the instruction window and instruction fetch/execute mechanisms of a conventional superscalar processor are inefficient, i.e. the lengths of critical paths do not scale well as the peak instruction bandwidth of the processor is increased. As a result, cycle time is potentially lengthened and the performance gains due to higher instruction throughput are diminished. Secondly, conventional superscalar processors are often unable to extract sufficient parallelism from sequential programs. This is due to *architectural limitations* in the management of control and data dependences that obscure instruction-level parallelism.

The purpose of this thesis is to explore a next generation microarchitecture, called the *trace processor* [112,96], that addresses both complexity and architectural limitations. The trace processor 1) exploits hierarchy and replication to manage the complexity of increasing peak hardware parallelism [80,39,112] and 2) performs sophisticated control and data flow management to partially alleviate dependence constraints on instruction-level parallelism [96,81,86].

1.1 Hierarchy and replication: managing complexity

The proposed microarchitecture (Figure 1-1) is organized entirely around *traces*. A trace is a long sequence of instructions, e.g. 16 to 32 instructions, that may contain any number of taken or not-taken branch instructions. Traces introduce hierarchy. Rather than working at the granularity of individual instructions, the processor more efficiently sequences through the program at the higher level of traces and allocates execution resources to trace-sized units of work.

As will be discussed in the following two subsections, the trace-based approach overcomes the basic inefficiencies of managing processor fetch and execution resources on an individual instruction basis.

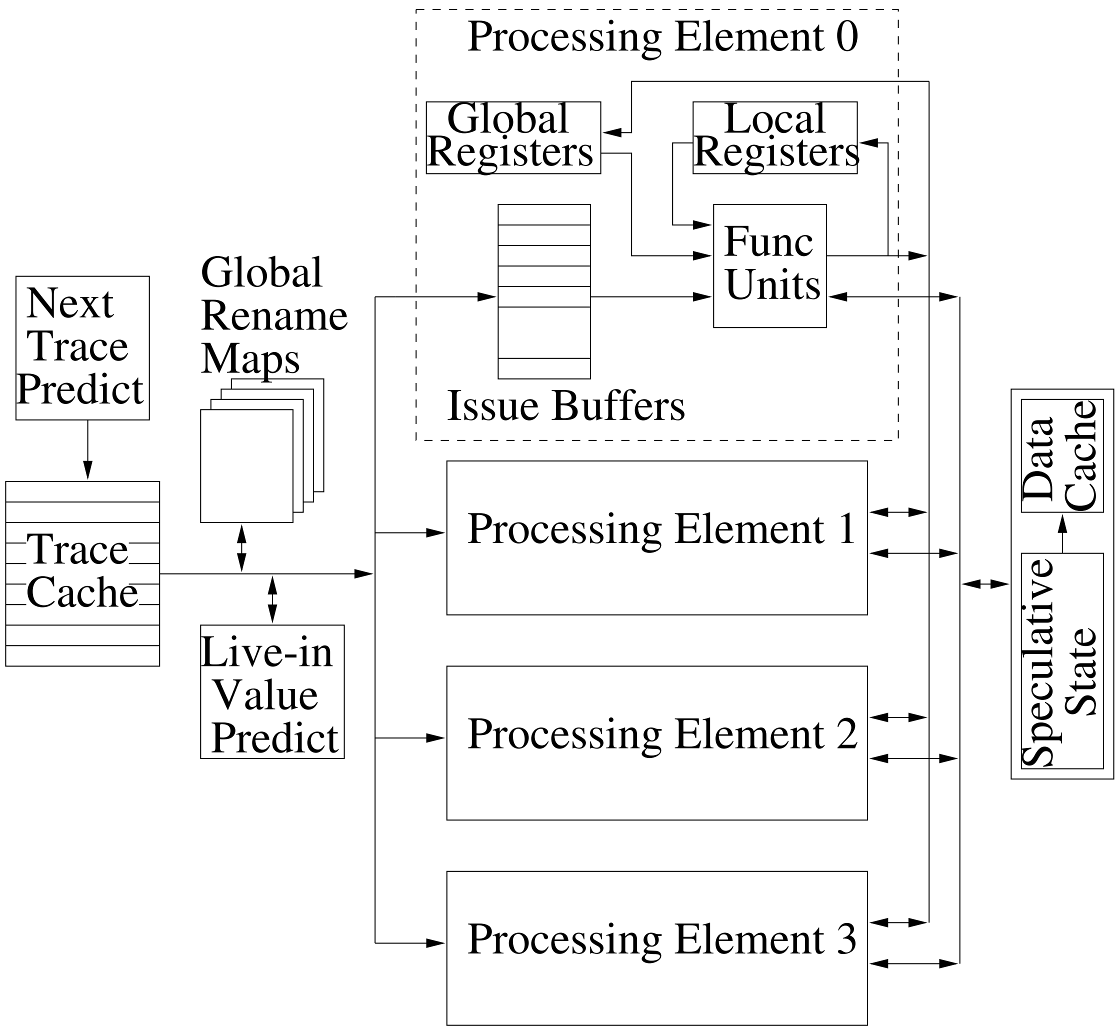


Figure 1-1: Trace processor.

1.1.1 Low latency, high bandwidth instruction fetching

As peak instruction issue rates grow from relatively modest, 4-way issue typical of today's processors to 8-way, 12-way, or even 16-way issue of future generation processors, it will become necessary to predict and fetch multiple basic blocks in a single cycle.

The trace processor achieves high bandwidth control flow prediction by treating traces as the fundamental unit of prediction, not individual branch instructions. A *next-trace predictor* [39] performs a single trace prediction and, in doing so, implicitly predicts multiple branches in a single cycle.

To greatly simplify the process of fetching multiple, possibly noncontiguous basic blocks in a single cycle, the instructions that form a trace are stored together as a single contiguous unit in a special instruction cache, called the *trace cache* [41,75,80,71]. A conventional instruction cache distributes instructions from the same trace among multiple, noncontiguous cache lines, and requires several cycles to assemble the trace. The distinction between a conventional instruction cache and the trace cache is shown in Figure 1-2.

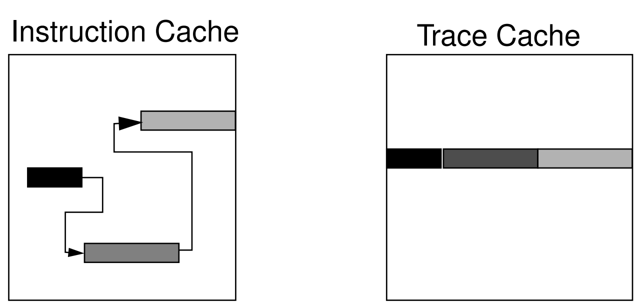


Figure 1-2: Distinction between conventional instruction cache and trace cache. A trace may contain any number of predicted-taken branches, in which case the instructions are stored in noncontiguous locations throughout a conventional instruction cache. The same group of instructions are stored together in a trace cache line.

1.1.2 A hierarchical instruction window organization

The instruction issue mechanism is possibly the most complex aspect of current dynamically scheduled superscalar processors [69]. Each cycle, the processor examines the instruction window for instructions that have received their input values and are ready to issue (*wakeup logic*). Of the ready instructions, a number of them are selected for issue based on issue bandwidth and other resource constraints (*select logic*). The selected instructions possibly read values from the register file and are routed to functional units, where they execute and write results to the register file. Each result must also be quickly bypassed to functional units to be consumed by pipelined, data dependent instructions (*operand bypass circuits*). Clearly, the four aspects of instruction issue -- wakeup logic, select logic, register file, and operand bypasses -- grow in complexity as the size of the instruction window, the instruction issue bandwidth, and number of parallel execution units are increased [69].

A trace is an “instruction window” itself, albeit a smaller one, and this realization can be exploited to break down complexity. A single trace window is depicted in Figure 1-3, with its data flow divided hierarchically into intra-trace and inter-trace values [112]. *Local values* are produced and consumed solely within a trace and are not visible to other traces. *Global values* are communicated among traces. Global input values to a trace are called *live-ins* and global output values of a trace are called *live-outs*.

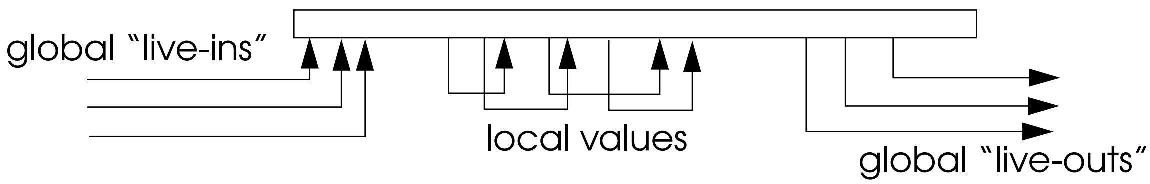


Figure 1-3: Data flow hierarchy of traces.

Based on this data flow hierarchy, the large trace processor instruction window is distributed among multiple smaller processing elements (PEs), as shown in Figure 1-1. Each PE resembles a small-scale superscalar processor and at any given time is allocated a single trace to process. A PE has 1) enough instruction issue buffers to hold an entire trace, 2) multiple dedicated functional units, 3) a dedicated local register file for storing local values, and 4) dedicated local result buses, corresponding to write ports into the local register file and local result bypasses.

Logically, a single global register file stores global values. Each PE contains a copy of the global register file for private read bandwidth. Write bandwidth to the global register file cannot be similarly distributed: all PEs are connected to a common set of global result buses, which provide the datapaths for bypassing global values among traces and writing values into the global register file.

The hierarchical instruction window simplifies instruction issue mechanisms.

- *Instruction wakeup logic* monitors fewer result tag buses. Although each PE monitors both its own local result tags and all global result tags, overall, fewer tags are monitored due to reduced global register traffic.

- *Instruction select logic* is fully distributed. Each PE independently selects ready instructions from its trace and routes them to dedicated functional units.
- *Register read/write* bandwidth is satisfied by private local register ports and the corresponding, reduced reliance on global register ports.

  The size and complexity of the global register file is reduced with respect to a conventional superscalar register file because much of the register traffic is off-loaded to the local register files. That is, for an equivalent instruction window, the global register file requires fewer registers and fewer read/write ports than the monolithic file of superscalar processors.
- *Fast bypassing* of local values among functional units within a PE is feasible, despite a longer latency for bypassing global values among PEs.

  The slower scaling of interconnect delay relative to logic delay has been identified as a serious problem in forthcoming IC technologies [6]. Result bypasses are primarily long interconnect and are impacted most by this trend [69]. The trace processor exploits communication locality in programs [21] to localize interconnect, most notably the local result bypasses within PEs; managing values uniformly would unnecessarily penalize communication latency for all values. A technique for partially hiding global communication latency is discussed in Section 1.2.1.1.

Traces must be dispatched to the PEs at a maximum rate of one per cycle as they are predicted and fetched by the trace processor frontend. Higher fetch bandwidth usually implies increasingly complex dependence checking and register renaming hardware. Fortunately, as with control flow prediction and instruction fetching, the instruction dispatch stage is simplified by dispatching a trace as a single unit.

The identification of local values, live-ins, and live-outs of a trace are performed at trace cache fill time, and the dependence information cached with the trace. This essentially moves the complex dependence checking logic from the dispatch stage to the fill side of the trace cache. Furthermore, local values are pre-renamed to the local register file during dependence checking of the trace [112].

Now, the dispatch stage operates efficiently at the trace-level, by virtue of renaming only live-in and live-out registers [112]. Live-in and live-out registers are renamed to the global register file to establish register dependences with other traces in the window. The trace dispatch stage is simplified in the following ways.

- Dependence checking logic is eliminated and live-in/live-out registers are explicitly labeled in the trace for fast renaming.
- Register rename map complexity is reduced because the checkpointing granularity is increased. Maps are checkpointed at trace boundaries instead of at every branch instruction.
- Merging instructions into the window is simplified. A single trace is routed to a single PE, whereas an instruction-granularity processor routes multiple instructions to as many, possibly noncontiguous instruction buffers.

1.1.3 Summary via analogy

Prior to superscalar processors, comparatively simple dynamically scheduled processors fetched, dispatched, issued, and executed one instruction per cycle, as shown in the left-hand side of Figure 1-4. The branch predictor predicts up to one branch each cycle and a single PC fetches one instruction from a simple instruction cache. The renaming mechanism, e.g. Tomasulo's algorithm [105], performs simple dependence checking by looking up a couple of source tags in the register file. And at most one instruction is steered to the reservation station of a functional unit each cycle. After completing, instructions arbitrate for a common data bus, and the winner writes its result and tag onto the bypass bus and into the register file.

The superscalar paradigm "widens" each of these pipeline stages in a manner that increases complexity with each additional instruction per cycle. This is clearly manageable up to a point: high-speed, dynamically scheduled 4-way superscalar processors currently set the standard in microprocessors. But there is a crossover point beyond which it becomes more efficient to manage instructions in groups, that is, hierarchically.

The trace processor is one possible approach for managing instructions hierarchically. In the right-hand side of Figure 1-4, the top-most level of the trace processor hierarchy is shown (the trace-level). The picture is virtually identical to that of the single-issue out-of-order processor on the left-hand side. The unit of operation has changed from 1 instruction to 1 trace, but the pipeline bandwidth remains 1 *unit* per cycle.

In essence, grouping instructions within traces is a reprieve. Complexity does not necessarily increase with each additional instruction added to a trace. Additional branches are absorbed, as are additional source and destination operands. Hardware parallelism is allowed to expand incrementally -- up to a point, at which time perhaps another level of hierarchy, and another reprieve, is needed.

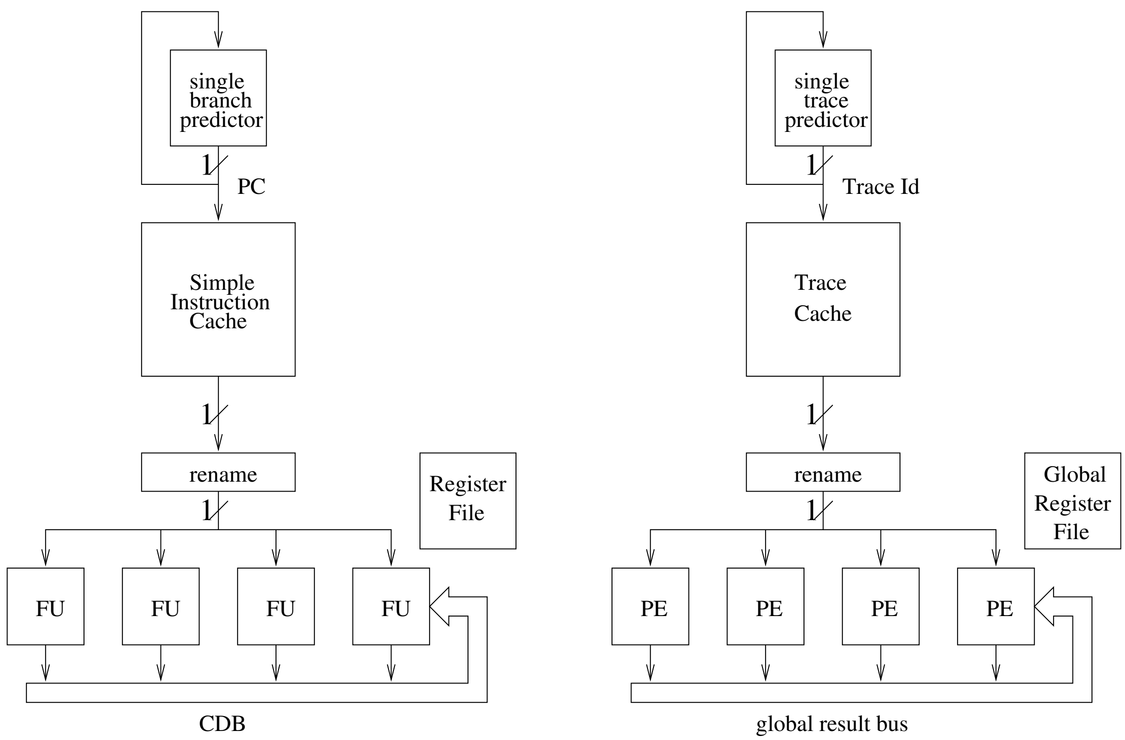


Figure 1-4: Analogy between a single instruction and a single trace.

1.2 Speculation: mitigating data and control dependences

Instruction-level parallelism is limited by data dependences between instructions. Pairs or groups of instructions form dependence chains; instructions within a chain execute serially but instructions among multiple, disjoint chains may execute in parallel. A large instruction window exposes this irregular parallelism, partially alleviating the effects of data dependences. However, fundamental architectural limitations remain.

1. An effective instruction window relies on accurate trace prediction. Even relatively infrequent trace mispredictions severely limit the size of the *useful* instruction window and are often catastrophic to performance.

2. Data dependences ultimately cause poor utilization of the peak issue bandwidth, even with 100%-accurate trace prediction. With longer interconnect delays imminent, the latency for resolving data dependences is likely to increase, worsening the serial nature of dependence chains.

3. A significant subset of data dependences are not immediately identifiable by the processor, namely, dependences between load and store instructions. Resolving memory dependences conservatively, i.e. assuming loads depend on all prior unresolved stores, only worsens the problem of true data dependences by introducing false ones.

The trace processor features advanced speculation techniques to partially address these problems. Section 1.2.1 describes two data speculation techniques, the first a novel application of value prediction [51,87,27] to break inter-trace data dependences and the second a variant of the address resolution buffer [24] for speculative memory disambiguation. Section 1.2.2 describes a sophisticated control flow management technique, *control independence* [46,20,83], for maintaining accurate instruction windows despite trace mispredictions.

1.2.1 Data speculation techniques

1.2.1.1 Live-in prediction

When a trace is renamed and dispatched to a PE, its live-in values are predicted (Figure 1-1). Predicting live-in values -- as opposed to predicting all trace values or performing no value prediction at all -- is meaningful in several ways.

Of course, predicting only inter-trace values meshes with the overall trace processor strategy of performing operations at the trace-level. Trace-level value prediction reduces bandwidth to the value prediction tables and working set of the value predictor.

The hierarchical trace processor, with its multiple processing elements, is in some sense a parallel processor. Correctly predicting live-ins decouples and enhances the parallel processing of traces and, in particular, improves *load balance*. A more sophisticated alternative would be to select traces carefully based on data dependences, i.e. data-dependence-based *trace selection* [115].

Finally, live-in values are inherently slower to compute due to global communication latency. Value prediction in this case is more critical to performance than if the same values were local. One can even make a new case for value prediction: it is potentially a key *enabler* for distributing the instruction window, clocking the chip very fast, and necessarily lengthening global latencies. This is purely conjecture, although Chapter 6 presents limited supporting data.

1.2.1.2 Memory dependence speculation

A load instruction issues to the data cache as soon as its address is available and a cache port is free. There may or may not be dependences with prior unresolved store instructions in the window. Thus, a simple memory dependence prediction is performed: predict no dependence [24].

If a load issues before a prior dependent store, then the prediction is incorrect and the load must re-issue to the cache to get the correct store value. Also, all subsequent data dependent instructions must be selectively re-issued.

A variant of the address resolution buffer (ARB) [24] is used to detect memory dependence violations and selectively re-issue incorrect load instructions. Selectively re-issuing all subsequent data dependent instructions is achieved transparently and automatically via the underlying selective recovery model of the trace processor, described in the next subsection.

1.2.1.3 Selective recovery model

Because data speculation is pervasive in the trace processor, data misspeculation is relatively frequent and recovery plays a central role in the design of the trace processor data dependence mechanisms. There are two challenges.

1. Recovery must be selective, i.e. only those instructions affected by incorrect data should be re-executed [51]. Misspeculation is frequent enough to demand high-performance recovery.

2. There are three sources of data misspeculation -- live-in mispredictions, misspeculated load instructions, and partially incorrect control flow (control independence, Section 1.2.2). Often, multiple sources of data misspeculation interact in complex ways. The number of unique recovery scenarios is virtually unlimited. All scenarios must be handled with a simple selective recovery model.

Selective recovery is divided into two steps. First, the “source” of a violation is detected and only that instruction re-issues. *Detection is performed by the violating instruction itself*. In this way, the detection logic is distributed; and since detection occurs at the source, it is trivially clear which instruction to re-execute. Detection mechanisms vary with the type of violation (mispredicted live-in, load violation, etc.) and are described in Chapter 3.

Second, *the existing instruction issue logic* provides the mechanism to selectively re-issue all incorrect-data dependent instructions. When the violating instruction re-issues, it places its new result onto a result bus. Data dependent instructions simply re-issue by virtue of receiving new values.

An instruction may issue any number of times while it resides in the instruction window. It is guaranteed to have issued for the final time when it is ready to retire (all prior instructions have retired), therefore, traces remain allocated to PEs until retirement.

1.2.2 Control independence

Branch instructions are a major obstacle to maintaining a large window of useful instructions because they introduce control dependences: the next group of instructions to be fetched following a branch instruction depends on the outcome of the branch. High performance processors deal with control dependences by using branch prediction. Predicting branch outcomes allows instruction fetching and speculative execution to proceed despite unresolved branches in the window. Unfortunately, branch mispredictions still occur, and current implementations squash all instructions after a mispredicted branch, thereby limiting the effective window size. Following a squash, the window is often empty and several cycles are required to re-fill it before instruction execution proceeds at full efficiency.

Often only a subset of dynamic instructions immediately following a branch truly depend on the branch outcome, however. These instructions are *control dependent* on the branch. Other instructions deeper in the window may be *control independent* of the mispredicted branch: they will be fetched regardless of the branch outcome, and do not necessarily have to be squashed and re-executed [46,20,83].

Control independence typically occurs when the two paths following a branch re-converge before the control independent instruction, as depicted in Figure 1-5. Upon detecting the misprediction, the incorrect control dependent instructions are squashed and replaced with the correct control dependent ones, but processing of control independent instructions can proceed relatively unaffected. Exploiting control independence requires three basic mechanisms outlined below.

1. The re-convergent point must be identified in order to distinguish and preserve the control independent instructions in the window.

2. The processor must support insertion and removal of control dependent instructions from the *middle* of the window.

3. The mispredicted control flow may cause some incorrect data dependences to be formed between control independent instructions and instructions before the re-convergent point. These incorrect data dependences must be repaired and the incorrect-data dependent, control independent instructions selectively re-executed.

The re-convergent point is where the incorrect and correct control dependent paths meet, and instructions after the re-convergent point are control independent instructions. This definition is different from a static definition of control independence. In a static definition, the re-convergent point is where *all possible control dependent paths* meet (often called the post-dominator basic block). Therefore, the statically-defined re-convergent point can not be closer to the mispredicted branch (in terms of dynamic instructions) than our dynamically-defined re-convergent point. In either case, control independence is exploited only if the re-convergent point is within the instruction window.

Control independence is an effective technique for mitigating the effects of branch mispredictions. A recent study shows potential performance improvements of 30% in wide-issue superscalar processors [83]. However, practical mechanisms for the three outlined requirements need to be explored. In [83], control dependence information is ideally conveyed from the compiler to hardware for identifying re-convergent points, yet simple hardware-only detection of re-convergent points is desirable. And the reorder buffer of superscalar processors is managed as a fifo with insertion and removal of instructions performed only at the head and tail of the fifo, not in the middle. Arbitrary expansion and contraction, beginning and ending at *any point* in the window, may be complex. Finally, conventional register and memory dependence mechanisms are inadequate for control independence. A new overall data flow management strategy is needed -- one that supports data speculation in general.

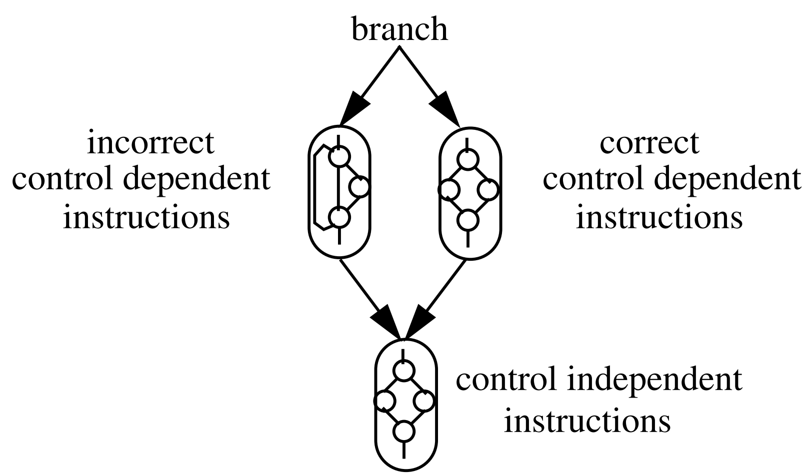


Figure 1-5: Control independence example.

In this thesis, the trace processor microarchitecture is explored as a practical and effective platform for control independence. There are three major contributions towards control independence.

1. Trace-level re-convergence is not guaranteed despite re-convergence at the instruction-level. Novel trace selection techniques are developed to expose control independence at the trace-level.

2. Control independence's potential complexity stems from insertion and removal of instructions from the middle of the instruction window. Trace processors manage control flow hierarchically (traces are the fundamental unit of control flow) and this results in an efficient implementation.

3. Control independent instructions must be inspected for incorrect data dependences caused by mispredicted control flow. Existing data speculation support is easily leveraged to selectively re-execute incorrect-data dependent, control independent instructions.

1.3 Thesis, contributions, and outline

The thesis of my research is that <u>trace processors are a good microarchitecture for implementing wide-issue machines</u>. I defend this thesis by arguing three key points.

1. *Trace processors are an evolutionary extension of superscalar processors*.

   Trace processors do not require instruction set architecture changes and, consequently, they maintain *binary compatibility*. Binary compatibility is arguably a major reason for the success of dynamic superscalar processors because it enables new processor generations to run existing software.

   Trace processors retain a *single flow of control*. Wide instruction fetching enables instruction dependences to be established quickly and, likewise, instructions to be scheduled quickly. This approach is *robust* in that it performs well over a range of applications. And it does not rely on sophisticated and potentially less-robust multithreading or very-long instruction word (VLIW) compilers to accurately schedule instruction fetching from multiple, disjoint points in the program.

2. *Trace processors demonstrate better overall performance than conventional superscalar counterparts.*

   Distributing resources results in less scheduling flexibility and non-uniform operand bypassing, both of which reduce the average number of instructions executed per cycle. But the cycle time of a distributed processor is more sensitive to *single PE complexity* and not the entire processor, which gives the trace processor an overall performance advantage over conventional superscalar processors. Overall, trace processors outperform aggressive superscalar counterparts because the trace processor microarchitecture enables both high ILP and a fast clock.

3. *The trace processor organization naturally supports aggressive speculation.*

   The contiguous instruction window enables aggressive, but relatively transparent, selective recovery from data misspeculation. Control flow hierarchy and existing data speculation support are leveraged to manage the complexity of exploiting control independence.

The trace processor architecture draws from a substantial and influential body of research. The multiscalar execution paradigm [20,22,100] and its adaptation from compiler-defined tasks to dynamic traces [112,103,96,81] lay solid foundations for the trace processor. These and other related work are discussed at length in Chapter 2 - “Related Work”.

My contributions are two-fold. First, I fully develop a trace processor microarchitecture in Chapter 3. There I describe *how* to build a processor organized around traces. Key components of a trace processor are identified and defined, and solutions are provided. The key components are 1) the trace cache based, hierarchical sequencing mechanism, 2) the distributed instruction window and hierarchical issue mechanisms, 3) data speculation mechanisms including selective recovery support, and 4) hierarchical control independence techniques.

Second, I present performance evaluations and complexity analysis that support the three key points listed above: the first two points (wide instruction fetching and overall performance advantage) are supported by experiments in Chapter 5 - “Evaluation of Hierarchy”, and the third point (platform for aggressive speculation) is supported by experiments in Chapter 6 - “Evaluation of Speculation”. My experimental method and simulator infrastructure are described in Chapter 4 - “Experimental Method”.

Chapter 2

Related Work

The organization of the related work discussion follows the overall organization of the thesis and trace processor: high bandwidth frontend, distributed execution model, data speculation, and control independence. As will be seen, my thesis leverages the contributions of an influential and highly relevant body of research.

2.1 High bandwidth instruction fetching

2.1.1 Alternative high bandwidth fetch mechanisms

At least four previous studies have focused on high bandwidth instruction fetching. All of these attempt to fetch multiple, possibly noncontiguous basic blocks each cycle from the instruction cache.

First, Yeh, Marr, and Patt [120] consider a fetch mechanism that provides high bandwidth by predicting multiple branch target addresses every cycle. The method features a *branch address cache*, which extends the branch target buffer (BTB) [47] to store a tree of target addresses. With a branch target buffer, a single branch prediction and a BTB hit produce the starting address of the next basic block. Similarly, a hit in the branch address cache combined with multiple branch predictions produces the starting addresses of the next *several* basic blocks. These addresses are fed into a highly interleaved instruction cache to fetch multiple basic blocks in a single cycle.

A second study by Dutta and Franklin [17] uses a similar approach to the branch address cache (providing multiple branch targets), but with a new method for predicting multiple branches in a single cycle. Their approach hides multiple individual branch predictions within a single prediction; e.g., rather than make 2 branch predictions, make 1 prediction that selects from among 4 paths. This enables the use of more accurate two-level predictors.

Another hardware scheme proposed by Conte, Mills, Menezes, and Patel [13] uses two passes through an interleaved branch target buffer. Each pass through the branch target buffer produces a fetch address, allowing two nonadjacent cache lines to be fetched. In addition, the interleaved branch target buffer enables detection of any number of branches in a cache line. In particular, the design is able to detect short forward branches within a line and eliminate instructions between the branch and its target using a *collapsing buffer*. The work also proposes compiler techniques to reduce the frequency of taken branches.

Seznec, Jourdan, Sainrat, and Michaud [91] propose a technique for achieving multiple branch prediction without tree-like predictor structures or sequential predictor accesses used in other approaches. The technique is called *multiple-block ahead prediction*. In conventional BTB sequencing, the current fetch block is used to lookup the immediate successor block, implying a serial chain of accesses to achieve multiple block prediction. To overcome this, the current fetch block can instead be used to predict some number of blocks ahead, breaking the serial access chain. This same prediction model can also be applied to pipelining the branch predictor in superpipelined processors, or for pipelining longer latency two-level predictors.

2.1.2 Trace cache development

Melvin, Shebanow, and Patt proposed the *fill unit* and *multinodeword cache* [58,59]. The first work qualitatively describes the performance implications of smaller or larger *atomic* units of work at the instruction-set architecture (ISA), compiler, and hardware levels. The authors argue for small compiler atomic units and large *execution atomic units* to achieve highest performance. The former is motivated by compile-time flexibility. Larger execution units are said to allow for more efficient processing: by using larger indivisible units of work, less processor state must be maintained for exception recovery. The fill unit is proposed as the hardware mechanism for compacting the smaller compiler units into the large execution units, which are then stored for reuse in a decoded instruction cache. The follow-on work [59] evaluates the performance potential of large execution atomic units. Although this work only evaluates sizes up to that of a single VAX instruction and a basic block, it also suggests joining two consecutive basic blocks if the intervening branch is "highly predictable".

In [60], software basic block enlargement is discussed. In the spirit of trace scheduling [19] and trace selection [35], the compiler uses profiling to identify candidate basic blocks for merging into a single execution atomic unit. The hardware sequences at the level of execution atomic units as created by the compiler. The advantage of this approach is the compiler can optimize and schedule across basic block boundaries.

Franklin and Smotherman [23] extended the fill unit's role to dynamically assemble VLIW-like instruction words from a RISC instruction stream, which are then stored in a *shadow cache*. This structure eases the issue complexity of a wide issue processor. They further applied the fill unit and a decoded instruction cache to improve the decoding performance of a complex instruction-set computer (CISC) [97]. In both cases, cache lines are augmented to store *trees* to improve the utilization of each line.

Four works have independently proposed the trace cache as a complexity-effective approach to high bandwidth instruction fetching. Johnson [41] proposed the *expansion cache*, which addresses cache alignment, branch prediction throughput, and instruction run merging. The expansion process also predetermines the execution schedule of instructions in a line. Unlike a pure VLIW cache, the schedule may consist of multiple cycles via *cycle tagging*. Peleg and Weiser [75] describe the design of a *dynamic flow instruction cache* which stores instructions independent of their virtual addresses, the defining characteristic of trace caches. Rotenberg, Bennett, and Smith [79,80] motivate the concept with comparisons to other high bandwidth fetch mechanisms (branch address cache and collapsing buffer, Section 2.1.1), both in terms of complexity and performance, and define some of the trace cache design space. Patel, Friendly, and Patt [71] expand upon and present detailed evaluations of this design space, arguing for a more prominent role of the trace cache. Two trace cache papers appear in a special issue on cache memories of the *Transactions on Computers* [73,84].

2.2 Processor paradigms

2.2.1 Multiscalar paradigm

2.2.1.1 Managing complexity in multiscalar processors

Work in the area of multiscalar processors, by Franklin and Sohi [20,21,22,24] and Sohi, Breach, and Vijaykumar [8,9,10,100,113,114,115], first recognized the complexity of implementing wide instruction issue in the context of centralized resources. They pointed out the difficulty of scaling instruction fetch and dispatch bandwidth, the register file, and the instruction window and associated issue mechanisms.

The result of their research is an interesting combination of compiler and hardware. The compiler divides a sequential program into *tasks*, each task containing arbitrary control flow. Task control flow boundaries are conveyed in the binary via instruction set extensions to guide the hardware sequencing mechanism.

The processor is comprised of multiple, replicated processing elements. At run-time, a hardware task-level sequencer predicts and schedules tasks onto the PEs. Task-level sequencing [100,38] is facilitated by a *task cache* for caching the binary's task information and a *next task predictor* that learns the history of prior task sequences to predict future task sequences. The PEs independently and concurrently fetch and execute instructions from their respective tasks. Therefore, the instruction fetch and prediction bottleneck is overcome by hierarchical sequencing.

- The task-level sequencer takes large strides through the dynamic instruction stream by predicting one task per cycle.

- Independent instruction-level sequencers among the PEs (i.e., replicated program counters, instruction caches, and branch predictors) provide *multiple flows of control* for implicit, high instruction fetch and branch prediction bandwidth.

A distributed register file and distributed issue mechanisms are proposed based on a study of register dependences [21]. The study reveals that register values have short lifetimes and are consumed fairly soon after being produced. Potentially many instances of an architected register are produced within a task, but only the *last* instance needs to be communicated to other tasks. Thus, there is a hierarchy of register values, those produced and consumed solely within a task and those communicated among tasks. The register hierarchy is exploited to distribute instruction issue and register dependence mechanisms.

- *Distributed register file*: Each PE has its own register file. Intra-task values (i.e. local values) are written to and read from only the physical register file allocated to the particular task. Other register files do not observe this register traffic.

  The PE register files are connected in a unidirectional ring (although an arbitrary *physical network* may be used to implement a *logical ring*) to communicate inter-task values among the distributed register files. When a task is dispatched to a PE, in addition to the start PC of the task, a *create mask* of all architected registers potentially written by the task is indicated. Task create masks are generated by the compiler, conveyed via the binary's task information, and eventually cached in the task cache. At run-time, create masks are forwarded to and accumulated by subsequent PEs, which then know to synchronize on the indicated registers, i.e., wait for prior tasks to communicate those registers. Special *forward bits* in instruction opcodes or explicit *register release instructions* indicate the last instance of an architected register within a task; the PE forwards the last instance to subsequent tasks via the inter-task communication ring.

- *Distributed issue mechanism*: Each PE forms its own instruction window for executing instructions within a task. The size of the PE window is in no way tied to the size of a task because tasks can be arbitrarily large. Instruction issue logic is greatly simplified in multiscalar processors because each PE subwindow is relatively small and each PE is provided dedicated functional units. Also, full operand bypassing of intra-task values is feasible, despite longer operand bypass delays for communicating inter-task values via the ring.

2.2.1.2 Aggressive speculation in multiscalar processors

The multiscalar compiler need not guarantee that tasks are control and data independent, a deviation from conventional parallelizing compilers. Thus, the hardware employs aggressive control and data speculation to concurrently execute tasks.

The problem of ambiguous memory dependences is exacerbated in multiscalar processors because loads and stores may be *fetched* out-of-order, a consequence of multiple flows of control. The *address resolution buffer* (ARB) [24] was proposed to allow load instructions to issue speculatively (always predict no dependence with prior tasks) and detect store-load dependence violations after the fact (i.e., when stores perform). Recovery involves squashing and restarting all subsequent tasks. To reduce the number of memory dependence squashes, explicit memory dependence prediction can be employed to synchronize loads and stores [63]; the ARB is still used to detect remaining mispredictions.

Register dependences are explicitly specified by the compiler via create masks. Create masks are typically used by the processor to synchronize tasks. Create masks are conservative, however, because they are a summary of registers modified along all paths through a task. Also, possible final register values that are control dependent on intra-task branches are typically forwarded to other tasks only after the intra-task branches are resolved. These are not fundamental restrictions and, as with memory dependences, register dependences may be speculated [113] given sufficient compiler and hardware support.

The multiscalar paradigm also promotes advanced control speculation. Control flow is managed hierarchically and this results in a conceptually simple implementation of control independence. A branch misprediction within a task does not necessarily cause subsequent tasks to squash if they are control independent of the mispredicted branch. Discussion of multiscalar control independence is deferred to Section 2.4, where control independence architectures are described at length.

2.2.1.3 Distributing the data cache and memory disambiguation hardware

This thesis does not consider the complexity of scaling data cache and memory disambiguation bandwidth. The *speculative versioning cache* [28] addresses the problem of load and store bandwidth. It features a distributed data cache that also performs memory renaming (versioning) and speculative memory disambiguation, for use in distributed processors and particularly the multiscalar paradigm. Each PE has its own data cache for private read/write bandwidth and a fast access time; furthermore, the data cache performs cache line versioning and detects speculative load mispredictions. A coherency-like mechanism ensures proper coordination of PE data caches. Related work along these lines includes the Stanford Hydra Project [31,32,65,66,67,68] and the CMU STAMPede Project [102], discussed in the next section.

2.2.2 Speculative multithreaded processors

I now briefly summarize speculative multithreaded processors [2,16,66,102,106] that have the following fundamental aspects in common with the multiscalar paradigm. This also provides an opportunity to summarize the potential architectural advantages of the multiscalar and speculative multithreading paradigm.

1. Threads (tasks) are extracted from a single, sequential program -- ordinary programs written in a sequential programming model. Thus, a simple programming model is retained and thread (task) information may be viewed as performance hints. Thread identification may be compiler or hardware oriented; the hardware thread selection work that I am aware of is [107,56,2].

2. Multiple flows of control are implemented to fetch and execute multiple threads concurrently and possibly out-of-order. There are primarily two advantages of multiple flows of control. Firstly, it enables large instruction windows and the extraction of distant parallelism without actually forming a single, *contiguous* instruction window. Secondly, multiple flows of control implies hierarchical management of control flow and, therefore, a conceptually simple implementation of control independence. A branch

misprediction within a thread does not necessarily cause subsequent threads to squash if they are control independent of the mispredicted branch (more on this in Section 2.4).

3. Threads are initiated speculatively, i.e., the compiler need not guarantee that parallel threads are control and data independent. The hardware provides mechanisms for run-time verification of speculation. Misspeculation occurs when there are data dependences among threads and the dependences are violated, or when control flow does not reach an initiated thread. Typical recovery actions involve squashing threads after and including the first misspeculated thread. The *dynamic multithreading architecture* (DMT) [2] buffers entire threads, however, enabling a certain degree of selective recovery: upon detecting misspeculation, entire threads are re-fetched from buffers and analyzed to isolate the affected instructions, and these instructions are selectively re-executed.

Projects that embody these architectural principles include the SPSM architecture [16], the Superthreaded architecture [106], Dynamic Multithreading [2], UPC Speculative Multithreading [54,55,56,107], and several single-chip multiprocessor projects -- the Stanford Hydra Project [31,32,65,66,67,68] and CMU STAMPede Project [102].

2.2.3 Trace processors

Traces are essentially “unwound”, dynamic versions of static multiscalar tasks (although tasks may be arbitrarily large). Vajapeyam and Mitra recognized the similarity of static tasks and dynamic traces, and proposed the first trace processor organization

[112,62]. In particular, they proposed 1) partitioning the instruction window based on traces, 2) partitioning the register file into a global file and per-trace local files, and 3) reusing register renaming information for registers local to a trace. Sundararaman and Franklin also recognized the similarity of tasks and traces, and proposed an adaptation of the multiscalar processor to trace-based execution [103].

Smith and Vajapeyam [96] put forth trace processors as a next generation architecture for exploiting forthcoming billion transistor chips. Rotenberg, Jacobson, Sazeides, and Smith [81] follow up on the trace processor proposal with a comprehensive microarchitecture definition and evaluation. The trace processor's aggressive control and data speculation techniques were initially described and explored in [81]. Jacobson, Rotenberg, and Smith [39] developed the next trace predictor. A full trace processor control independence microarchitecture and evaluation is presented in [86].

Traces can be pre-processed to speed their execution within PEs or, alternatively, reduce the strength of PEs without degrading performance [95,40]. Nair and Hopkins [64] employ *dynamic instruction formatting* to pre-schedule traces, although this work is suggested in the context of VLIW processors and not trace processors. Friendly, Patel, and Patt [26] and Jacobson and Smith [40] present a suite of dynamic trace optimizations, including explicit scheduling or assignment of scheduling priorities, collapsing of dependent operations for execution on combined functional units [87], and constant propagation.

2.2.4 Clustering based on data dependences

Ranganathan and Franklin [77] describe a taxonomy of decentralized ILP execution models. The instruction window may be decentralized based on 1) execution units, 2) control dependences, and 3) data dependences. One of the earliest dynamically scheduled processors, the Tomasulo-based IBM 360/91 [105], is an example of decentralization based on execution units -- a cluster of reservation stations is associated with a particular functional unit -- although only part of the scheduling mechanism (the logic that arbitrates among ready-to-issue reservation stations) is distributed. Examples of the second approach are multiscalar and trace processors, because tasks and traces are contiguous instruction sequences.

The PEWs architecture by Kemp, Ranganathan, and Franklin [43,78], dependence-based superscalar processors by Palacharla, Jouppi, and Smith [70], Multicluster architecture by Farkas, Chow, Jouppi, and Vranesic [18], and the DEC Alpha 21264 microprocessor [48] steer instructions to clusters of functional units (PEs) based on where their data dependences are likely to be resolved. The underlying idea is to minimize global communication among functional unit clusters and, conversely, maximize local communication within a cluster. Thus, data dependent instructions are likely to reside within the same cluster.

Data dependence clustering simplifies the instruction window in two ways. Firstly, instruction scheduling is simplified within a cluster. Instructions tend to form serial dependence chains in each queue, and perhaps only the instruction at the head of the queue needs to be considered for issue. Secondly, partial bypasses perform comparably to complete bypasses because communication is localized as much as possible within each cluster.

2.2.5 VLIW and block-structured ISAs

The concept of traces has long existed in the software realm of instruction-level parallelism. Early work by Fisher [19], Hwu and Chang [35], and others on trace scheduling and trace selection for microcode recognized the problem imposed by branches on code optimization. Subsequent VLIW architectures and novel ISA techniques, for example [36,61,33], further promote the ability to schedule long sequences of instructions containing multiple branches.

2.3 Data speculation

2.3.1 Value prediction

Lipasti, Wilkerson, and Shen observed the *value locality* property of programs [49,50,51], i.e. the property that many instructions produce and consume a small number of values and that these values are often predictable. Source or destination operands are predicted, and instructions execute speculatively based on the predictions. This results in higher instruction-level parallelism because dependent instructions that would otherwise execute serially now execute in parallel [50].

Sazeides and Smith [88] study a range of value predictors (constant, stride, and context-based). Their proposed context-based value predictors [88] provide the highest accuracy; this class of value predictor is used for live-in prediction in Chapter 6.

Smith and Vajapeyam [96] first proposed applying value prediction to inter-trace (inter-thread) dependences, and the follow-on work explores this venue [81]. Subsequently, at least two other architectures have proposed breaking inter-thread dependences using value prediction -- DMT [2] and the UPC speculative multithreaded architecture [107].

2.3.2 Memory dependence prediction

Franklin and Sohi [24] proposed a hardware approach for speculative memory disambiguation. In multiscalar processors, loads issue speculatively as soon as their addresses are available. The ARB tracks all speculatively-performed load instructions. When a store is performed, the ARB checks if any subsequent loads to the same address were speculatively performed; if so, the task containing the least-speculative dependent load is restarted and subsequent tasks are squashed.

As mentioned in Section 2.2.2, all speculative multithreaded processors likewise speculate on memory dependences and provide some form of run-time misspeculation detection. Recovery actions may be hardware or software based.

In trace processors, the ARB is modified to track only stores. The ARB creates multiple store versions and orders versions based on sequence numbers. Loads are still *serviced* by the ARB: the ARB returns the assumed-correct version of data based on sequence number comparisons. However, speculative loads are *tracked* by the PEs containing them. Misspeculation detection is therefore performed by the PEs, by monitoring stores as they issue on the cache buses. Moving the speculative load tracking function into the PEs is motivated by the selective recovery model. The trace processor's ARB variant was first published in [81].

Explicit memory dependence prediction may be used to significantly reduce the number of misspeculated load instructions. Moshovos, Breach, Vijaykumar, and Sohi [63] developed a highly accurate mechanism for predicting memory dependences and synchronizing dependent memory operations.

2.3.3 Selective re-issuing

Lipasti [51] first proposed selective re-issuing in the context of data speculation. Sazeides [89] formalized the concept by dividing the underlying mechanism into three steps, and proposed a taxonomy of speculative architectures based on these steps. The three steps are 1) check predictions (*equality*), 2) recover in the case of a value misspeculation (*invalidation*), and 3) inform direct and indirect successors of a correctly predicted instruction that their operands are valid (*verification*). According to [89], the superspeculative architecture [51] uses *parallel invalidation* and *parallel verification* -- that is, special hardware is required to quickly propagate invalidation and verification information to all direct and indirect successor (dependent) instructions.

To the best of my knowledge, the trace processor as proposed in [81] and in this thesis is one of the first *serial invalidation* and *serial verification* architectures, which has the advantage of exploiting the underlying issue and retirement mechanisms for performing invalidation and verification, respectively [89]. Invalidation is performed by virtue of receiving a new source operand value (issue mechanism), and verification is performed by virtue of the trace processor retirement model (instructions remain in their issue buffers until retirement). This thesis also contributes to selective re-issuing by articulating the conceptually complicated, but actually simple, interaction of multiple and diverse mispredictions (Section 1.2.1.3).

In addition to an in-depth analysis of data speculation support, Sazeides develops a detailed data-speculative microarchitecture [89] (a case study). The microarchitecture extends an RUU-based superscalar processor [99] to support parallel invalidation and verification.

Forms of selective recovery appear in at least three other contexts. Firstly, although entire tasks are squashed, the multiscalar processor selectively repairs registers in a well-orchestrated effort among the multiple, distributed register files [8]. Secondly, although all instructions are squashed after a branch misprediction, the instruction reuse buffer [98] selectively re-executes instructions based on the state of the reuse buffer. Finally, the DMT architecture [2] performs selective re-execution; instructions are re-fetched from the thread (buffered in the second-level window) and dependences are analyzed to find the minimum subset of instructions that require re-execution.

2.4 Control independence and other control flow techniques

2.4.1 Limit studies

Lam and Wilson's limit study [46] demonstrates that control independence exposes a large amount of instruction-level parallelism, on the order of 10 to 100, for control-intensive integer benchmarks. This work was followed up by Uht and Sindagi [111] in their study of "minimal control dependences" and showed similar results.

2.4.2 Control independence in multiscalar and multithreaded processors

Control independence is a property of a dynamically executed program. Ways of exploiting control independence can vary with the hardware and software techniques being used. I identify two general classes of implementations (although hybrids are possible).

- *Multiple flows of control with a noncontiguous instruction window.* This class of machines has multiple instruction fetch units and can simultaneously fetch from disjoint points in the dynamic instruction stream. The instruction window, i.e. the set of instructions simultaneously being considered for issue and execution, does not have to be a contiguous block from the dynamic instruction stream. Clearly, control independent code regions are good candidates for parallel fetching, though this is not a requirement. Multiscalar and speculative multithreaded processors fall into this class.
- *Single flow of control with a contiguous instruction window.* This class of machines has a single program counter and can fetch along a single flow of control at any given time. The instruction window is a contiguous set of dynamic instructions. Control independence is implemented by allowing the program counter to skip back and forth in the dynamic instruction stream. Superscalar processors fall into this class [82].

Each class of machines has advantages. With implementations having multiple flows of control, there is a natural hierarchical structure: each flow of control fetches and operates on its own "task" or thread. Control decisions are separated into inter-task and intra-task levels. Intra-task mispredictions can be isolated to the task containing the misprediction, and later control independent tasks can proceed in a fairly straightforward manner. This hierarchical task-based structure leads to what is effectively a non-contiguous instruction window where instructions can be fairly easily inserted and removed as control mispredictions occur. The hierarchy also allows for multiple branch mispredictions to be serviced simultaneously if they are in different tasks.

An advantage of a single control flow implementation is that the single fetch unit can scan all the instructions as it builds the single instruction window and, therefore, has more complete knowledge of potential dependences. This potentially leads to more aggressive data dependence resolution and recovery mechanisms (discussed below). In addition, these methods may be able to take advantage of finer grain control independence, at the level of individual basic blocks, for example.

Trace processor control independence is a combination of the two models. A single flow of control is retained for a more complete picture of potential dependences, but control flow is managed hierarchically to simplify instruction window management.

The aggressive data dependence resolution and recovery mechanisms presented in this thesis bear important distinctions with other control independence architectures. Specifically, multithreading approaches may resolve inter-thread data dependences conservatively [113]. That is, even though control flow within a thread does not directly affect other threads, values dependent on the control flow may not be forwarded to other threads until the control flow is resolved. If speculative data forwarding is performed, entire threads are squashed when incorrect values are referenced, losing some or all of the benefits of control independence [103,113]. This is only true for designs without selective reissuing capability, e.g. large threads may preclude being selective. In a sense, this approach to control independence more closely resembles guarding [3,76], which converts control dependences into data dependences. This is not always the case: Vijaykumar, Breach, and Sohi [113] study several register forwarding strategies, including speculative register forwarding guided by (compiler) profiling to reduce the frequency of task squashes.

The more recently proposed *dynamic multithreading architecture* (DMT) [2] buffers entire threads in a second-level instruction window, enabling a certain degree of selective recovery. Upon detecting a mispredicted branch, entire threads are re-fetched from buffers and analyzed to isolate data dependent, control independent instructions, and these instructions are selectively re-executed.

2.4.3 Control independence in superscalar processors

Rotenberg, Jacobson, and Smith [82,83] examine the potential of control independence in the context of wide-issue superscalar processors. An aggressive implementation achieves improvements on the order of 30%. The proposed mechanisms are complex due to the non-hierarchical superscalar organization, and there is a reliance on the compiler to provide complete control dependence information. The study in [83] is intended to bound the performance potential of control independence and determine if it is worthwhile to explore practical implementations, and is therefore a necessary prelude to my trace processor research effort.

The instruction reuse buffer by Sodani and Sohi [98] provides another way of exploiting control independence. It saves instruction input and output operands in a buffer -- recurring inputs can be used to index the buffer and determine the matching output. In the proposed superscalar processor with instruction reuse, there is complete squashing after a branch is mispredicted. However, control independent instructions after the squash can be quickly re-evaluated via the reuse buffer.

Another approach by Chou, Fung, and Shen [12] uses a dual reorder buffer design. A misprediction causes one of the reorder buffers to complete squash, but control and data independent instructions from the second reorder buffer are preserved.

2.4.4 Control independence in trace processors

Control independence in trace processors is first introduced by Rotenberg, Jacobson, Sazeides, and Smith [81] but, because it is not the focus of that paper, the major issues are not formalized, conveyed, nor fully understood. The problem is not formalized in terms of *fine-grain control independence* (FGCI) and *coarse-grain control independence (*CGCI), and trace-level reconvergence and its implications to trace selection are not discussed. (FGCI and CGCI are two types of control flow management: FGCI refers to branches whose control dependent paths fit within a trace and CGCI refers to branches whose control dependent paths span multiple traces.) Performance gains are observed in two of the benchmarks, and these gains are due to manually-inserted, FGCI-like trace selection hints conveyed in the benchmark binaries; PEs are managed in a fifo queue so CGCI is not explicitly exploited. A more sophisticated treatment of trace processor control independence is presented in [86], and is the direct basis for the control independence concepts presented in this thesis.

2.4.5 Predication and multi-path execution

Predication [3,76,53,4] and selective multi-path execution [111,34,110,44,116,1] attempt to identify hard-to-predict branches, either through profiling or branch confidence estimators (respectively), and fetch both paths of these branches. In the case of multi-path execution, both paths are fully renamed and executed as separate threads. When the branch is resolved, one of the threads is squashed and the other becomes the primary thread of execution.

Predication is in some sense the software equivalent of multi-path execution applied to forward-branching regions of the CFG. In one form of predication, the control dependent instructions do not execute until their predicates are computed, i.e. multiple paths are fetched but only the correct path is executed. Alternatively, with *predicate promotion* [104] or *predicated state buffering* [4], instructions from multiple paths may execute concurrently, and only the results from the correct path are committed.

Predication and multi-path execution waste resources by fetching and possibly executing both the correct and incorrect control dependent paths of branches. This results in a performance gain over conventional speculation if the branches are mispredicted. Unfortunately, multi-path execution is applied to some fraction of correctly predicted branches, and alternatively, some fraction of incorrectly predicted branches are not covered by multi-path execution.

A problem specific to predication is the aggravation of data dependences. The purpose of branch prediction is two-fold: 1) quickly determine which instructions to fetch next and 2) quickly establish and resolve data dependences among instructions. Predication only addresses the first aspect. It “removes” branches, so the instructions to be fetched are known in advance (all instructions in the predicated region are fetched). It does not, however, address the second aspect. Without predicated state buffering, all predicated instructions must wait for their controlling predicate to be resolved. Branch prediction eliminates this control dependence if the prediction is correct, and it is correct more often than incorrect. With predicated state buffering, instructions within a region need not wait for predicates, but their computed results are not forwarded *outside* the region until predicate conditions are resolved.

The idea behind control independence is to always trust branch prediction and speculation, and take measures only when a misprediction occurs, thereby avoiding the above difficulties. When a misprediction does occur, predication and multi-path execution can potentially reduce the branch misprediction penalty more than control independence, because only part (or none) of the path after the branch is recovered in the case of control independence. On the other hand, because only a single path is followed, control independence may still capture more control independent instructions within the window than predication or multi-path execution.

Dynamic Hammock Predication (DP) [45] is loosely related to the FGCI technique developed in Section 3.4.3, only in that both techniques exploit `if-then` and `if-then-else` constructs. There are clear distinctions between my trace selection work and DP. First, FGCI and any form of control independence is not predication. FGCI relies on branch prediction and reduces the misprediction penalty when mispredictions do occur. Predication eagerly executes multiple control dependent paths in anticipation of mispredictions. Second, the DP technique is not fully dynamic -- the compiler identifies and marks `if-then` and `if-then-else` regions for predication. FGCI on the other hand implements *dynamic detection* of control flow constructs. Lastly, my FGCI algorithm can dynamically analyze arbitrarily complex, nested forward-branching code, whereas DP is restricted to regions containing only a single conditional branch.

2.4.6 Other misprediction tolerance techniques

The *mispredict recovery cache* proposed by Bondi, Nanda, and Dutta [7] caches instruction threads from alternate paths of mispredicted branches. The goal of this work is to quickly bypass the multiple fetch and decode stages of a long CISC pipeline following a branch misprediction.

Friendly, Patel, and Patt [25] propose *inactive issue*, a misprediction recovery model based on the trace cache. During trace cache indexing, if only a prefix of a trace matches the branch predictions, the non-matching instructions are still dispatched into the instruction window, where they are inactive. If the branch predictions for the trace were incorrect, and in fact the fetched trace is the correct path, the inactive instructions are quickly activated to reduce the misprediction recovery penalty.

Chapter 3

Trace Processor Microarchitecture

The trace processor microarchitecture is developed in this chapter. Figure 3-1 shows the key processor components that are covered, and in which sections they are covered. Section 3.1 describes the trace processor frontend. Section 3.2 covers the distributed instruction window: trace dispatch (Section 3.2.1), PE resource management (Section 3.2.2), and the processing element (Section 3.2.3). Section 3.3 describes the data speculation mechanisms.

Finally, control independence is described in Section 3.4. Control independence involves trace selection (Section 3.4.1.2), PE resource management (Section 3.4.1.1), and data speculation support (Section 3.4.1.3).

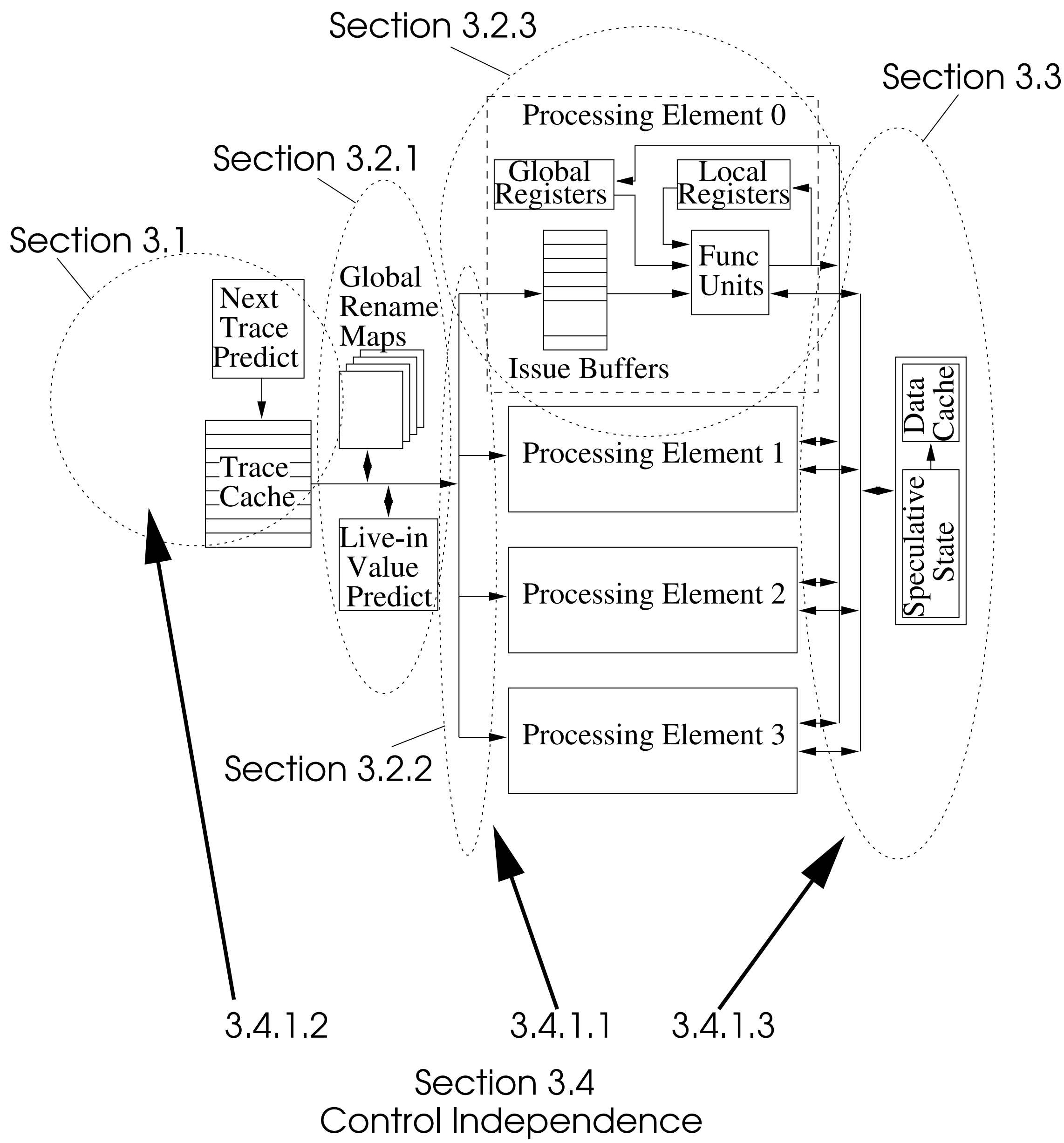


Figure 3-1: Roadmap for covering the trace processor microarchitecture.

3.1 Trace processor frontend

The trace processor frontend, shown in Figure 3-2, is designed to provide high instruction fetch bandwidth with low latency. This is achieved by explicitly sequencing through the program at the higher level of traces, both for 1) control flow prediction and 2) fetching instructions.

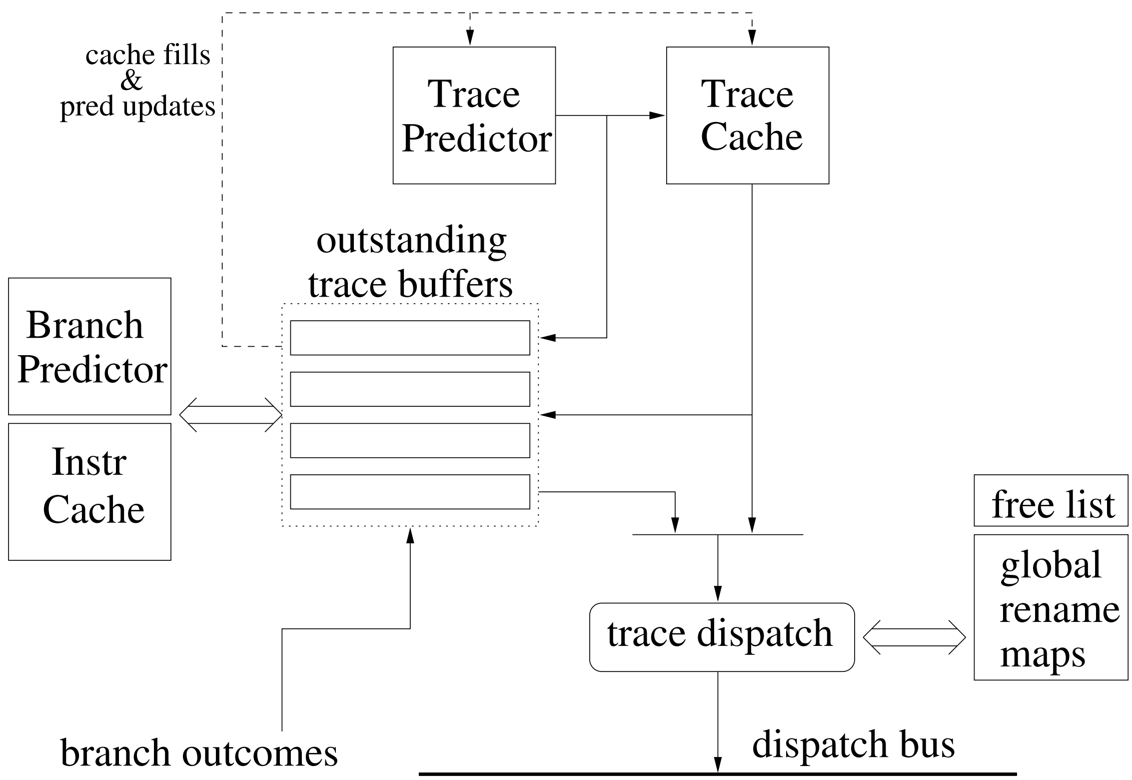


Figure 3-2: Trace processor frontend.

3.1.1 Trace-level sequencing

The trace predictor and trace cache together provide fast trace-level sequencing. The output of the trace predictor is a *trace identifier*: a given trace is uniquely identified by its starting PC and the outcomes of all conditional branches embedded in the trace. The trace identifier is used to lookup the trace in the trace cache. The index into the trace cache can be derived from just the starting PC, or a combination of PC and branch outcomes. Using branch outcomes in the index has the advantage of providing *path associativity* -- multiple traces emanating from the same starting PC can reside simultaneously in the trace cache even if it is direct mapped [80].

The output of the trace cache is one or more traces, depending on the cache associativity. A trace identifier is stored with each trace in order to determine a trace cache hit, analogous to the tag of conventional caches. The desired trace is present in the cache if one of the cached trace identifiers matches the predicted trace identifier.

Ideally, during trace-level sequencing, the next trace is predicted correctly and it hits in the trace cache. The trace is passed to the dispatch stage where live-in and live-out registers are renamed, establishing the register dependences with previous traces in the processor. The renamed trace is allocated to a PE via the dispatch bus.

Unfortunately, trace-level sequencing does not always provide the required trace. Instruction-level sequencing, discussed in the next subsection, is required to construct non-existent traces or repair trace mispredictions.

3.1.2 Instruction-level sequencing

The *outstanding trace buffers* in Figure 3-2 are used to 1) construct new traces that are not in the trace cache and 2) track branch outcomes as they become available from the execution engine, allowing detection of mispredictions and repair of the traces containing them. There is one outstanding trace buffer per PE.

Each fetched trace is dispatched to both a PE and its corresponding trace buffer. In the case of a trace cache miss, only the trace prediction is received by the allocated buffer. The trace prediction (starting PC and branch outcomes) provides enough information to construct the trace from the instruction cache, although this typically requires multiple cycles due to predicted-taken branches. Meanwhile, the trace dispatch pipe is stalled -- no other traces may pass through renaming because of the missing trace. However, the fetch stage is free to continue predicting traces, and these traces are placed in their outstanding trace buffers despite not reaching the dispatch stage. When the missing trace has been constructed and pre-renamed, the dispatch pipe is restarted and supplied with traces from the buffers in predicted program order. The non-blocking fetch pipe allows multiple trace cache misses to be serviced in parallel, restricted only by the number of datapaths to/from the instruction cache.

In the case of a trace cache hit, the trace is dispatched to the buffer. This allows repair of a partially mispredicted trace, i.e. when a branch outcome returned from execution does not match the path indicated within the trace. In the event of a branch misprediction, the trace buffer begins re-constructing the tail of the trace (or all of the trace if the start PC is incorrect) using the corrected branch target and the instruction cache. For subsequent branches in the trace, a *second-level branch predictor* is used to make predictions.

When a trace buffer is through constructing and pre-renaming its trace, it is written into the trace cache and dispatched to the execution engine. If the newly constructed trace is a result of misprediction recovery, the trace identifier is also sent to the trace predictor for repairing its path history.

3.1.3 Trace prediction

The trace predictor, shown in Figure 3-3, is based on Jacobson's work on path-based, high-level control flow prediction [38,39].

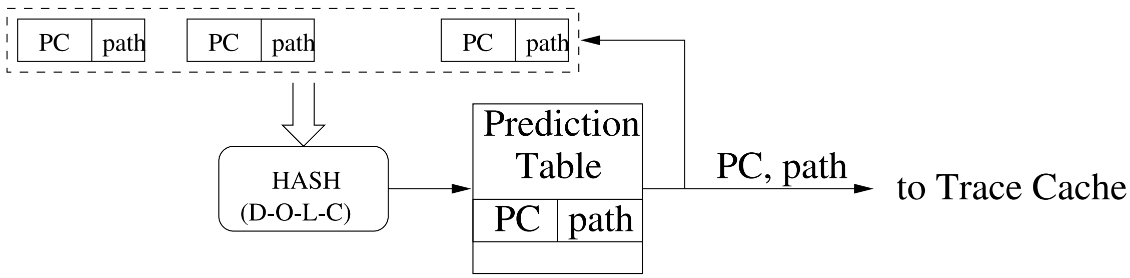


Figure 3-3: Jacobson's trace predictor.

An index into a correlated prediction table is formed from the sequence of past trace identifiers. The hash function used to generate the index is called a **DOLC** function: **'D'**epth specifies the path history depth in terms of traces; **'O'**ldest indicates the number of bits selected from each trace identifier except the two most recent ones; **'L'**ast and **'C'**urrent indicate the number of bits selected from the second-most recent and most recent trace identifiers, respectively.

Each entry in the correlated prediction table contains a trace identifier and a 2-bit counter for replacement. The predictor is augmented with several other mechanisms [39].

- *Hybrid prediction*. In addition to the correlated table, a second, smaller table is indexed with only the most recent trace identifier. This second table requires a shorter learning time and suffers less aliasing pressure.

- *Return history stack*. At call instructions, the path history is pushed onto a special stack. When the corresponding return point is reached, path history before the call is restored. This improves accuracy because control flow following a subroutine is highly correlated with control flow before the call.

- *Alternate trace identifier*. An entry in the correlated table is augmented with an alternate trace prediction, a form of associativity in the predictor. If a trace misprediction is detected, the outstanding trace buffer responsible for repairing the trace can use the alternate prediction *if it is consistent with known branch outcomes in the trace*. If so, the trace buffer does not have to resort to the second-level branch predictor; instruction-level sequencing is avoided altogether if the alternate trace also hits in the trace cache.

3.1.4 Trace selection

Trace processor performance is strongly dependent on *trace selection*, the algorithm used to divide the dynamic instruction stream into traces. At a minimum, trace selection is composed of a few simple trace termination rules, dictated by basic hardware constraints. The most notable of these is maximum trace length, i.e. trace cache line size. Trace line size is perhaps the most important selection constraint because it affects the performance of all trace processor components and usually influences other, more sophisticated trace selection constraints.

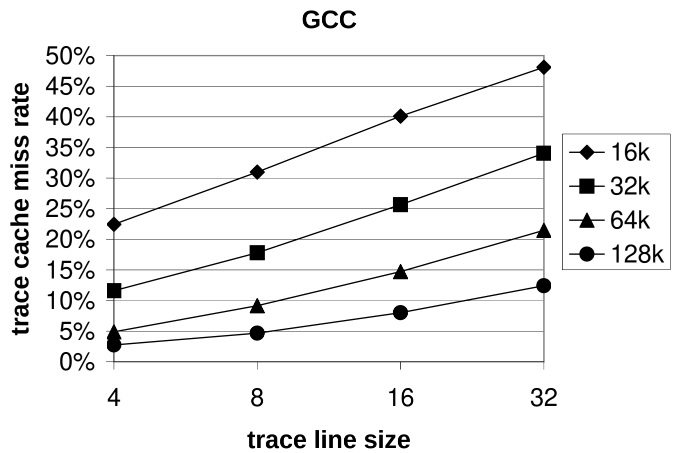
In addition to basic termination rules, trace selection may include sophisticated heuristics designed to improve various aspects of the trace processor. Trace selection is not comprehensively investigated in this thesis and it remains an important area of future research. Nonetheless, implications of trace selection are discussed below.

- *Trace cache performance*

  Trace line size affects *bandwidth per cache hit* and *cache hit rate*, both of which factor into overall fetch bandwidth.

  The interaction between trace line size and hit rate, however, is difficult to assess because it depends on the quality of trace selection. Trace cache miss rate versus trace line size is plotted for two benchmarks in Figure 3-4, using only basic trace selection (stop at the maximum trace length or at any indirect control transfer instruction). Four trace cache configurations are represented, 16KB through 128KB, all 4-way associative.

  For a fixed-size trace cache, miss rate increases with longer trace lines. Less constrained trace selection results in more static to dynamic expansion and, consequently, more traces. Furthermore, there are fewer total trace lines in the cache to accommodate the line size increase. Finally, additional trace selection constraints (specifically, stopping at indirect control transfers) create traces shorter than a trace line: internal fragmentation tends to increase with trace line size. All of these factors combine to increase the trace cache miss rate.



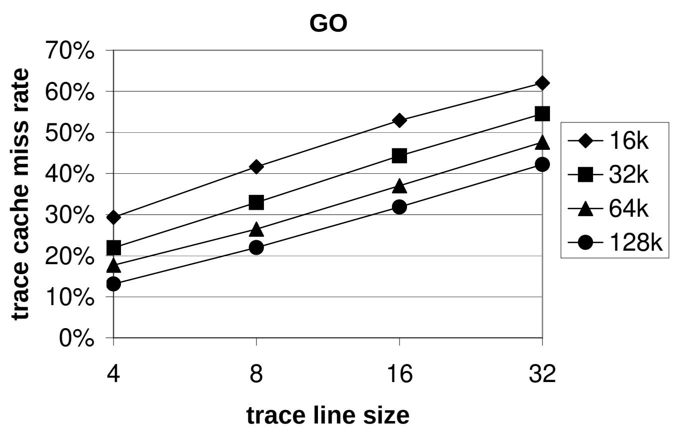


Figure 3-4: Impact of trace line size on trace cache miss rate.

Sophisticated selection techniques that are conscious of control flow constructs -- loops in particular -- may lead to different conclusions. The reader is referred to [71,81,72] for a few interesting control-flow-conscious selection heuristics.

- *Trace predictor performance*

Preliminary results indicate trace predictor performance is quite sensitive to trace selection. As observed in Section 5.1.2, the effect of more constrained trace selection is to create shorter and fewer unique traces. Firstly, shorter traces contain fewer branch instructions, so a given DOLC function captures less global path history. Secondly, fewer unique traces naturally decreases the amount of unique context for making predictions. Both of these effects combine to reduce trace prediction accuracy.

- *Trace construction latency*

  The latency of trace construction increases with trace length and, furthermore, a trace must be fully constructed before restarting the trace dispatch pipeline. Therefore, any benefits due to longer trace lines must be weighed against the higher trace construction penalty.

- *Hierarchical instruction window*

  Trace line size affects the complexity and performance of the hierarchical instruction window. A longer trace line increases the ratio of local values to global values. Therefore, complexity may be shifted away from global resources to the processing element.

  Performance implications are less clear. With longer traces, more communication is localized, potentially reducing the latency of dependence chains. But load balance among PEs is also affected. Whether longer traces improve or degrade load balance is unknown. This aspect of trace selection is investigated in Chapter 5.

  In the future, sophisticated heuristics might manage load balance based on 1) critical path estimates within traces and 2) data dependences among traces (see [114,115]).

- *Enhancing parallel trace execution*

  Trace selection can possibly analyze data flow to enhance the parallel execution of traces. Parallel trace execution is enhanced explicitly by reducing inter-trace dependences and scheduling the remaining dependences (see [114,115]), or implicitly by exposing highly-predictable values at trace boundaries.

- *Control independence*

  Trace selection gathers information about re-convergent points in the instruction stream to expose control independence. This aspect of trace selection is investigated in depth in Section 3.4.

3.1.5 Discussion of hierarchical sequencing

In Figure 3-5(a), a portion of the dynamic instruction stream is shown with a solid horizontal arrow from left to right. The stream is divided into traces T1 through T5. This sequence of traces is produced independently of where the instructions come from -- trace predictor/trace cache, trace predictor/instruction cache, or branch predictor/instruction cache.

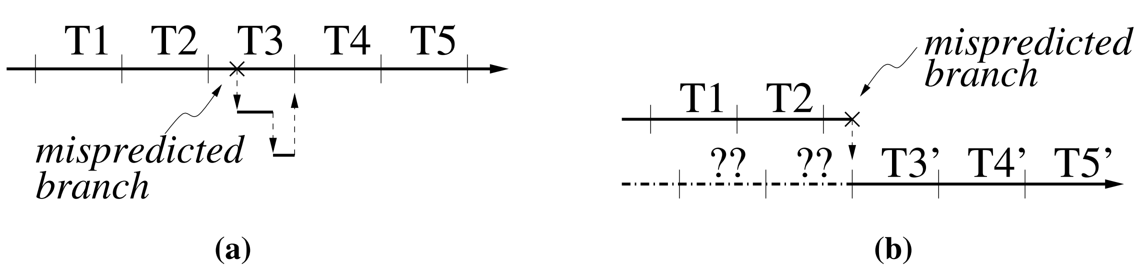


Figure 3-5: Two sequencing models. (a) Hierarchical (b) Non-hierarchical.

For example, if the trace predictor mispredicts T3, the trace buffer assigned to T3 resorts to instruction-level sequencing. This is shown in the diagram as a series of steps, depicting smaller blocks fetched from the instruction cache. The trace buffer strictly adheres to the boundary between T3 and T4, dictated by trace selection, even if the final instruction cache fetch produces a larger block of sequential instructions than is needed by T3 itself.

I call this process *hierarchical sequencing* because there exists a clear distinction between inter-trace control flow and intra-trace control flow. Inter-trace control flow, i.e. trace boundaries, is effectively *pre-determined* by trace selection and is unaffected by dynamic effects such as trace cache misses and mispredictions.

A contrasting sequencing model is shown in Figure 3-5(b). In this model, trace selection is “reset” at the point of the mispredicted branch, producing the shifted traces T3’, T4’, and T5’. This sequencing model does not work well with path-based next trace prediction. After resolving the branch misprediction, trace T3’ and subsequent traces must somehow be predicted. However, this requires a sequence of traces leading to T3’ and no such sequence is available (indicated with question marks in the diagram).

A potential problem with hierarchical sequencing is misprediction recovery latency. Explicit trace prediction uses a level of indirection: a trace is first predicted, and then the trace cache is accessed. This implies an extra cycle is added to the latency of misprediction recovery. However, this extra cycle is not exposed. First, consider the case in which the alternate trace prediction is used. The primary and alternate predictions are supplied by the trace predictor at the same time, and stored together in the trace buffer. Therefore, the alternate prediction is immediately available for accessing the trace cache when the misprediction is detected. Second, if the alternate is not used, then the second-level branch predictor and instruction cache are used to fetch instructions from the correct path. In this case, the instruction cache is accessed immediately with the correct branch target PC returned by the execution engine.

Unfortunately, the trace dispatch model aggravates both trace misprediction and trace cache miss recovery latency because a trace must be fully constructed before it is globally renamed and dispatched to a PE.

3.2 Distributed instruction window

3.2.1 Trace dispatch and retirement

The dispatch stage performs decode, global renaming, and live-in value predictions. Live-in registers of the trace are renamed by looking up physical registers in the most recent global register rename map. Independently, live-out registers receive new names from the free-list of global physical registers. The most recent global register rename map is copied (checkpointed) and the new map is updated to reflect the renamed live-out registers. The dispatch stage also looks up value predictions for all live-in registers.

The mechanism for committing register state is essentially the same as used by contemporary superscalar processors [94], only performed at the trace-level. The dispatch and retire stages manage a *trace-based active list* (or reorder buffer). At trace dispatch time, the trace is allocated an entry in the active list. An entry contains a list of *old physical registers* to which each live-out register was renamed prior to renaming this trace. At retirement, this list indicates which physical registers to return to the global freelist. Also included in the active list entry is a pointer to the register map used to rename the trace's live-in registers; this map is freed at retirement.

Precise exception handling is straightforward and conventional, but the trace model introduces a few unique aspects. If an exception is raised within a trace, processing of the exception is delayed until 1) the trace is at the head of the active list and 2) all instructions in the trace before the excepting instruction have completed execution. Then, the exception is handled like a branch misprediction, in that the corresponding outstanding trace buffer is notified of the exception, the register rename maps are backed up to the first map, and all PEs are squashed. The trace buffer backs up to the instruction just prior the excepting instruction, *terminates the trace at that instruction*, and pre-renames the "short trace" as it normally does during trace construction. The "short trace" is re-dispatched to a PE, it executes, and retires. At this point, state is precise and the exception handler is invoked.

3.2.2 Allocating and freeing PEs

Because the trace-based active list handles trace *retirement*, a PE can theoretically be freed as soon as its trace has *completed* execution. Unfortunately, knowing when a trace is "completed" is not simple, due to the waves-of-computation model: a mechanism is needed to determine when an instruction has issued for the last time [51].

A simpler approach is used in the trace processor. A PE is freed when its trace is retired because retirement guarantees instructions are done. The approach is somewhat costlier -- in terms of performance or resources, depending on the viewpoint -- because

PEs are freed in the same order they are allocated (i.e. FIFO), even though they might in fact complete out-of-order.

3.2.3 Processing element

The datapath for a processing element is shown in Figure 3-6. There are enough instruction buffers to hold the longest trace. For loads and stores, the address generation part is treated as an instruction in these buffers. The memory access part of loads and stores are placed into load/store buffers, which interface directly to cache ports (address and data buses). A set of registers close to the global register file stores live-in value predictions. Predictions are also dispatched to a separate validation unit which compares predictions to values received from other traces.

The timing of instruction issue is critical. The following two subsections describe local and global wakeup timings, respectively.

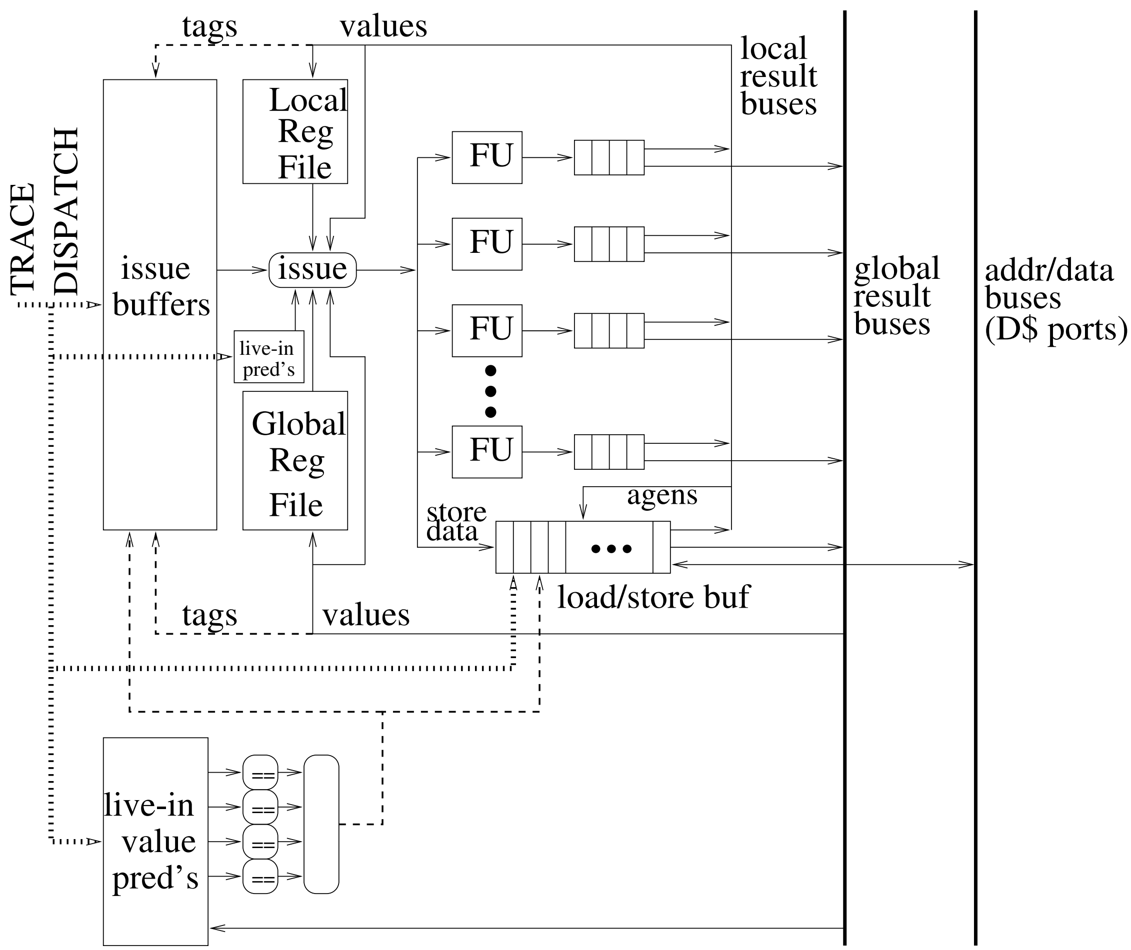


Figure 3-6: Processing element detail.

3.2.3.1 Local wakeup timing

The issue mechanism must ensure the arrival of an instruction at a functional unit coincides with the arrival of its most recently produced data operand(s). That is, critical data bypassing should dictate the timing of instruction execution. This is shown in Figure 3-7, where a locally-produced result can be bypassed in the same cycle it is produced and, therefore, dependent instructions execute in back-to-back cycles.

It takes one or more cycles to issue an instruction to a functional unit, e.g. in Figure 3-7, one cycle to wakeup and select the instruction and one cycle to obtain values from the register files. Therefore, to ensure the dependent instruction reaches a functional unit in time for the bypassed data, result *tags* must be broadcast in advance of the result *data*. In particular, as shown in Figure 3-7, there is a critical timing window -- just at the end of instruction wakeup/select -- during which the just-issued instruction broadcasts its tag in order to wakeup the dependent instruction in the following cycle.

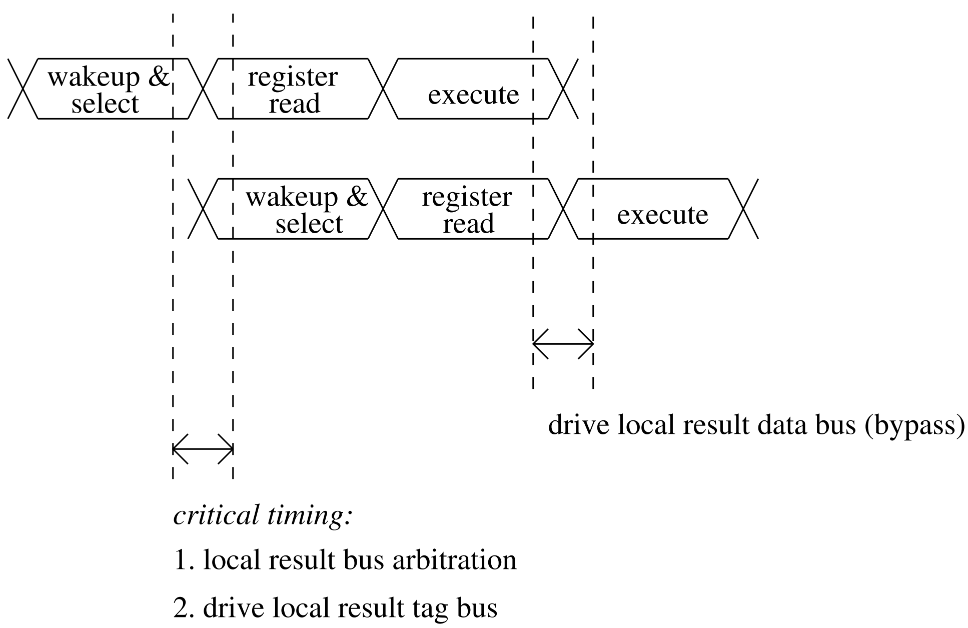


Figure 3-7: Local wakeup timing.

The process requires separate tag and data buses, as shown in Figure 3-8. Complicating matters is the need to arbitrate for a local result bus. A local bus arbiter maintains a schedule of free and used bus slots (a sophisticated version of the schedule in [93]). The timing window for back-to-back wakeup, described above, is even more critical because a local result bus slot must be obtained before broadcasting the tag. The critical logic cycle in Figure 3-8 -- from instruction window, to local bus arbiter, to local result tag bus, back to instruction window -- is characteristic of any small-scale, out-of-order superscalar processor.

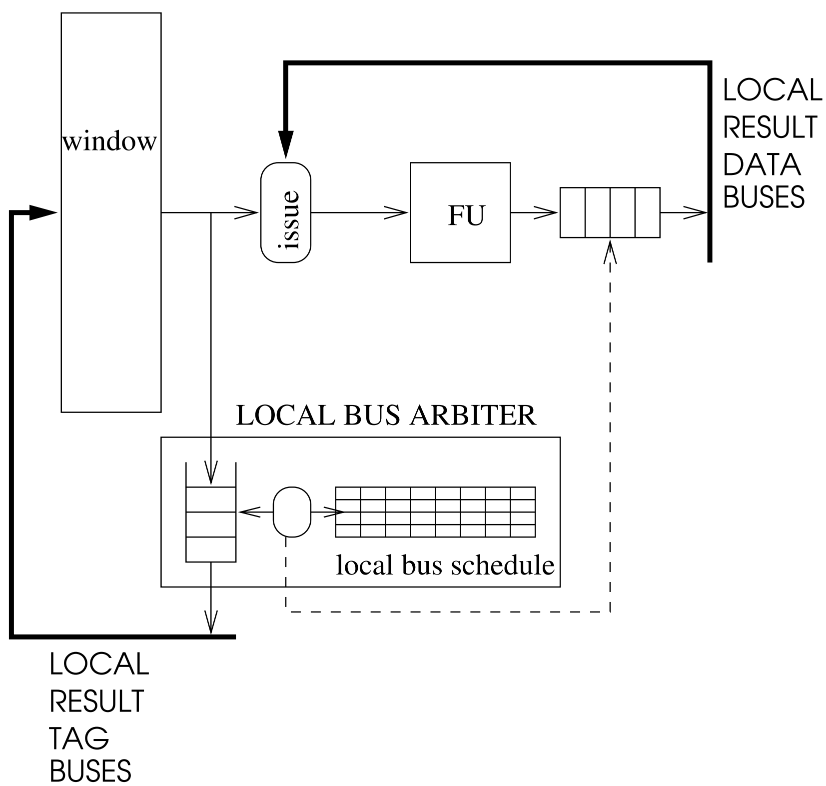


Figure 3-8: Local wakeup datapath and control.

Issue is not delayed if a local result bus is unavailable, although conventional processors typically delay issue. The trace processor is somewhat unique in that a result may be consumed both locally and globally (i.e. it is renamed to both local and global register files). If either a local or global result bus is available, but not both, then I do not want to stall issue.

Because issuing proceeds regardless of result bus availability, additional buffering and tracking is required for issued instructions. Associated with each functional unit is a buffer for holding completed results. For control, information about outstanding instructions is queued within the local bus arbiter. The local bus arbiter 1) arbitrates on behalf of outstanding instructions awaiting a local result bus, 2) drives the local result tag buses, and 3) sends signals to the functional unit buffers to direct data onto the local result data buses.

3.2.3.2 Global wakeup timing

The timings (Figure 3-9) and datapaths (Figure 3-10) for global wakeup are similar to that of local wakeup: there are separate global tag and data result buses; tags are broadcast in advance of the data; global result buses are arbitrated (completely independent of local arbitration); buffering and tracking are required for outstanding instructions awaiting a global result bus.

There are two major differences, however. First, it takes longer to broadcast global tags and data due to cross-PE communication latency, e.g. one extra cycle in Figure 3-9. This means dependent instructions in different PEs cannot execute back-to-back. However, the issue mechanism still ensures the timeliest schedule as dictated by the minimum global communication latency.

Second, arbitration is slightly more timing-critical because it requires communication among all PE arbiters (the wires labeled “global arbitration” in Figure 3-10). Timely arbitration is achieved by a combination of factors: 1) point-to-point interconnect, 2) only a small amount of readily-available information needs to be communicated among PEs (e.g.

number of buses requested), and 3) it may be feasible to "cycle-steal" a fraction of the subsequent tag broadcast cycle or preceding wakeup/select cycle. Alternatively, another cycle can be added to the global bypass latency (2 cycles instead of 1) to account for global bus arbitration.

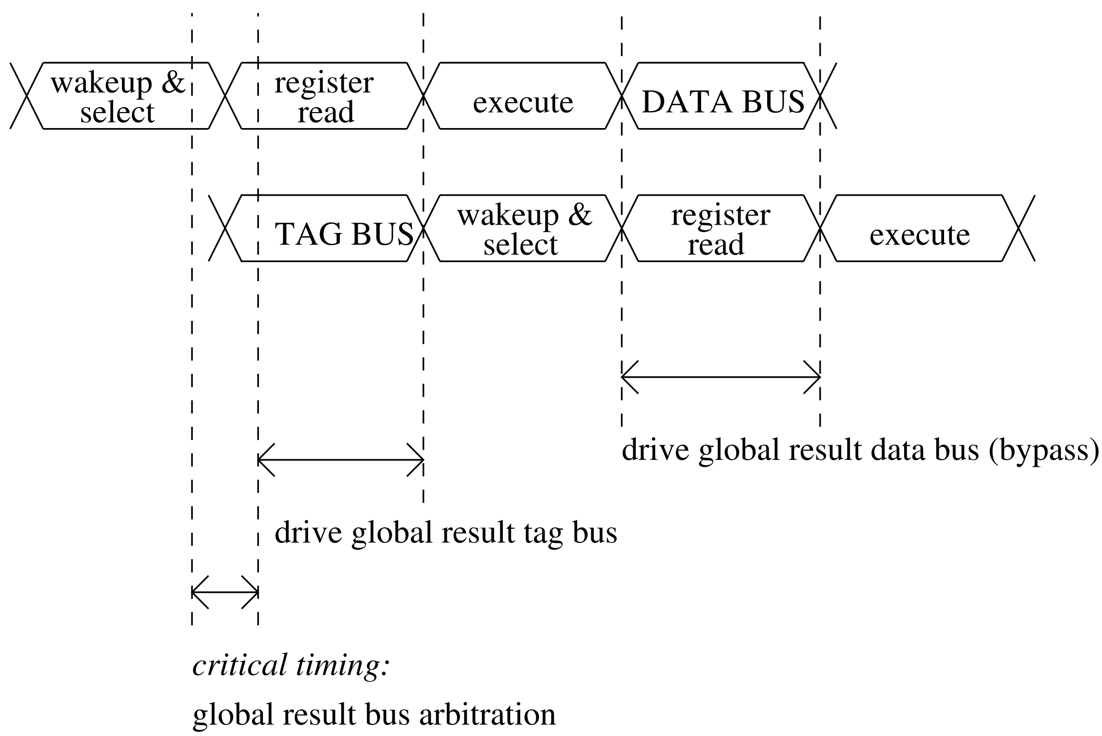


Figure 3-9: Global wakeup timing.

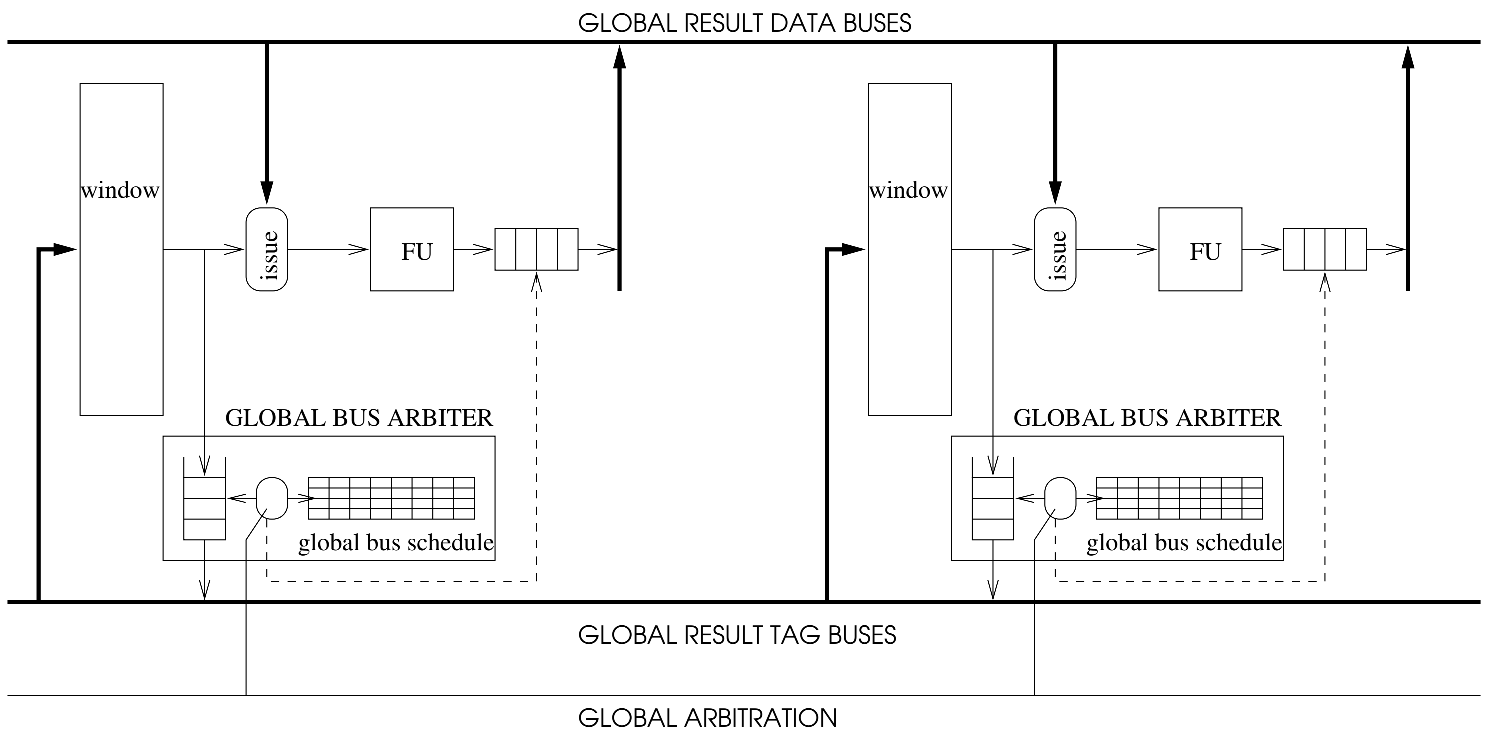


Figure 3-10: Global wakeup datapath and control.

3.3 Data misspeculation

Because data speculation is pervasive in the trace processor, data misspeculation is relatively frequent and recovery plays a central role in the design of the trace processor data dependence mechanisms. There are two challenges.

1. Recovery must be selective, i.e. only those instructions affected by incorrect data should be re-executed [51]. Misspeculation is frequent enough to demand high-performance recovery.
2. There are three sources of data misspeculation -- live-in mispredictions, misspeculated load instructions, and partially incorrect control flow (control independence, Section 3.4). Often, multiple sources of data misspeculation interact in complex ways (an example is given below). The number of unique recovery scenarios is virtually unlimited. All scenarios must be handled with a simple selective recovery model.

Selective recovery is divided into two steps. First, the “source” of a violation is detected and only that instruction re-issues. *Detection is performed by the violating instruction itself*. In this way, the detection logic is distributed; and since detection occurs at the source, it is trivially clear which instruction to re-execute. Detection mechanisms vary with the type of violation (mispredicted live-in, load violation, etc.) and are described in the subsections that follow.

Second, *the existing instruction issue logic* provides the mechanism to selectively re-issue all incorrect-data dependent instructions. When the violating instruction re-issues, it places its new result onto a result bus. Data dependent instructions simply re-issue by virtue of receiving new values.

An instruction may issue any number of times while it resides in the instruction window. It is guaranteed to have issued for the final time when it is ready to retire (all prior instructions have retired), therefore, traces remain allocated to PEs until retirement.

An example scenario is shown in Figure 3-11. Five instructions are shown from left to right, and four snapshots in time show execution progress. Arrows indicate dependence chains: dashed arrows are unresolved dependences, solid arrows are incorrectly resolved dependences, and thick solid arrows are correctly resolved dependences. There are initially three predictions: two incorrect live-in predictions for instructions i1 and i4 and a correct memory dependence prediction for the load instruction i3. The number of times each instruction has issued is indicated.

- Snapshot 1: Instruction i1 issues with an incorrectly predicted value but its result has not yet propagated to the address generation of store i2. The load i3 has issued but its result has not yet propagated to instruction i4. Instruction i4 has issued with an incorrectly predicted value and, in turn, caused i5 to issue.

- Snapshot 2: The store i2 computes an incorrect address but has not yet issued. The load i3 propagates a correct value to i4 and i4 detects its initial prediction was incorrect; i4 re-issues, as does i5. At this point, the dependence chain i3-i4-i5 is correctly but temporarily resolved.

- Snapshot 3: The store i2 incorrectly issues to address A. The load i3 detects a violation (even though it is actually correct) and re-issues itself. The new, incorrect load value gets propagated through instructions i4 and i5.

- Snapshot 4: Instruction i1 receives its input value and detects that its initial prediction was incorrect, so i1 re-issues; the store i2 generates a correct address B and then issues to that address. The load i3 detects that it was incorrectly linked to the store i2 and re-issues, in turn causing instructions i4 and i5 to re-issue.

Instructions issue up to four times in this example. Several “waves of computation” are required to arrive at the final, correct result.

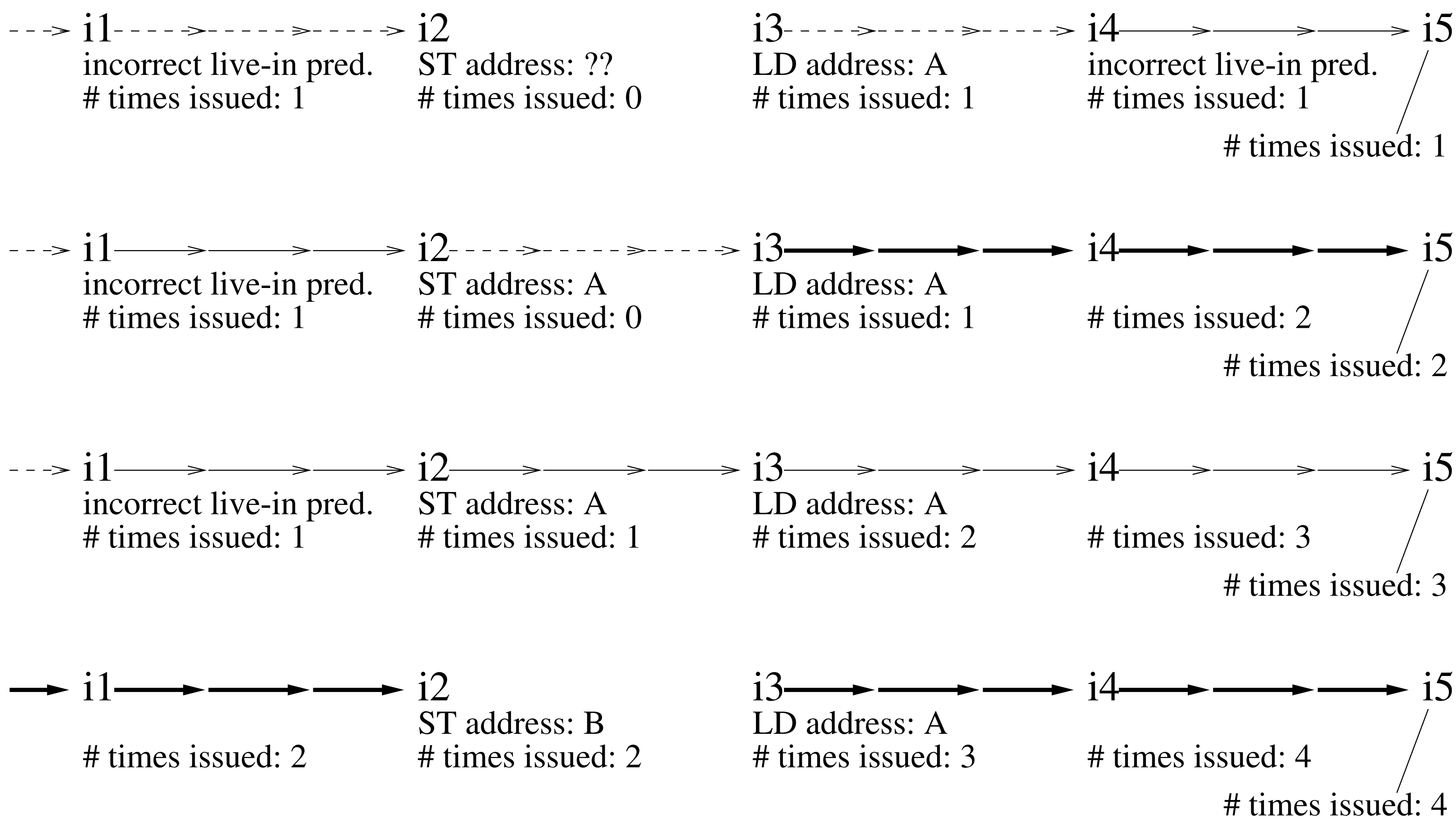


Figure 3-11: A complex scenario involving the selective recovery model.

3.3.1 Live-in value mispredictions

Live-in predictions are validated when the computed values are received from the global result buses. Instruction buffers and store buffers monitor comparator outputs corresponding to live-in predictions they used (Figure 3-6). If the predicted and computed values match, instructions that used the predicted live-in are not re-issued. Otherwise they do re-issue, in which case the validation latency appears as a misprediction penalty, because in the absence of speculation the instructions may have issued sooner [51].

3.3.2 Memory dependence mispredictions

The memory system (Figure 3-12) is composed of a data cache, an ARB derivative [24], distributed load/store buffers in the PEs, and memory buses connecting them. The ARB buffers speculative memory state and maintains multiple versions per memory location (memory renaming).

When a trace is dispatched, all of its loads and stores are assigned *sequence numbers*. Sequence numbers indicate the program order of all memory operations in the window. Handling of stores and loads are discussed in each of the next two subsections.

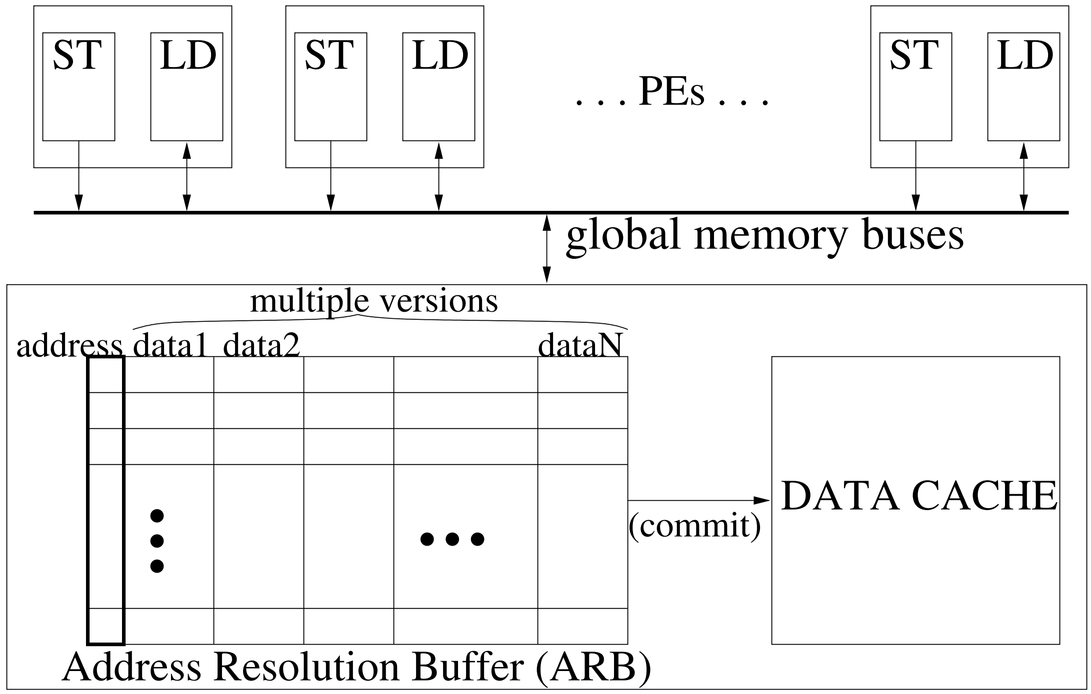


Figure 3-12: Abstraction of the memory system.

3.3.2.1 Handling stores

- When a store first issues to memory, it supplies its address, sequence number, and data on one of the memory buses. The ARB creates a new version for that memory address and buffers the data. Multiple versions are ordered via store sequence numbers.

- If a store must re-issue because it has received a new computed address, it must first "undo" its state at the old address, and then perform the store to the new address. Both transactions are initiated by the store sending its old address, new address, sequence number, and data on one of the memory buses.

- If a store must re-issue because it has received new data, it simply performs again to the same address.

3.3.2.2 Handling loads

- A load sends its address and sequence number to the memory system. If multiple versions of the location exist, the memory system knows which version to return by comparing sequence numbers. The load is supplied both the data and the sequence number of the store which created the version. Thus, *loads maintain two sequence numbers*: its own and that of the data.

- If a load must re-issue because it has received a new computed address, it simply re-issues to the memory system as before with the new address.

- Loads snoop all store traffic (store address and sequence number). A load must re-issue if 1) the store address matches the load address, 2) the store sequence number is logically less than that of the load (i.e. the store is before the load in program order), and 3) the store sequence number is logically greater than that of the load data (i.e. the load has an incorrect, older version of the data). This is a true memory dependence violation.

  The load must also re-issue if the store sequence number simply matches the sequence number of the load data. This takes care of the store 1) sending out new data or 2) changing its address via a store undo operation (a false dependence had existed between the store and load).

3.3.2.3 Regarding sequence numbers

The sequence numbers assigned to loads and stores are derived simply from the PE number plus location in the PE's trace, and are called *physical sequence numbers*. To perform comparisons, however, the physical sequence numbers must first be translated to *logical sequence numbers* based on the logical order of PEs.

Essentially, a small table maps physical PE number to logical PE number. This functionality rests with trace processor window management, so additional details regarding sequence number translation is deferred to Section 3.4.2.

3.4 Control independence

3.4.1 Overview

An overview of trace processor control independence is presented in the following three subsections. I begin with the hierarchical instruction window and how it is inherently suited to flexible window management. Then, the interesting problem of ensuring and identifying *trace-level re-convergence* is described. Finally, I describe how the selective misspeculation recovery model of trace processors supports the data flow management requirements of control independence.

3.4.1.1 Exploiting hierarchy: flexible window management

The hierarchical instruction window enables flexible window management in two ways.

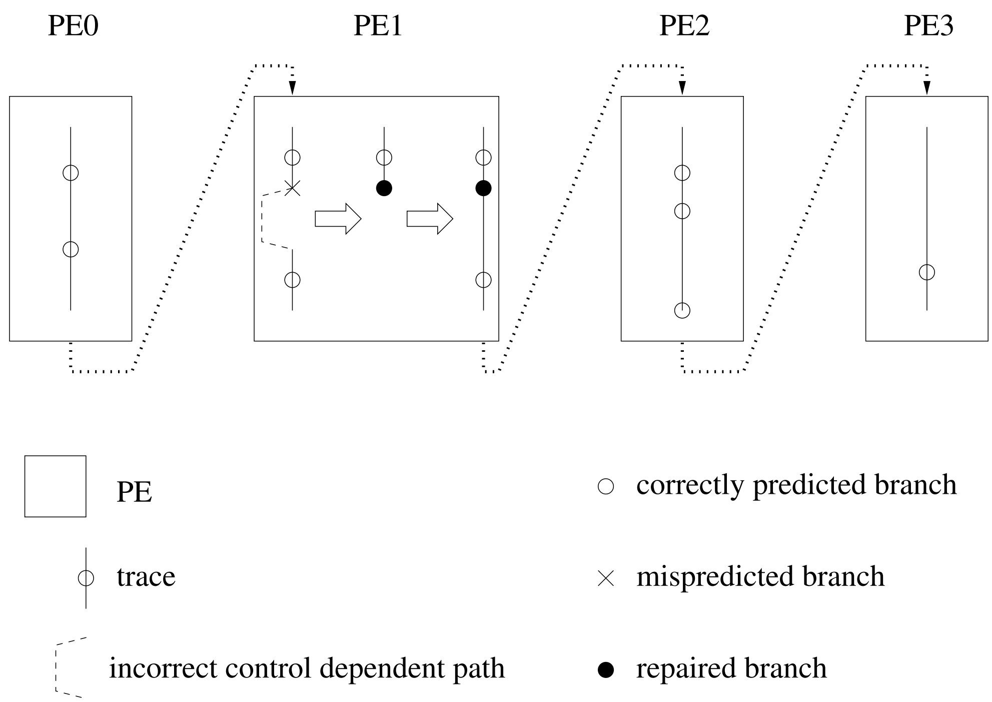
1. *Hierarchical management of control flow.* In some cases it is possible to isolate the effects of *intra-trace control flow* from *inter-trace control flow.* This is true if the longest control dependent path of a branch fits entirely within a trace. If such a branch is mispredicted, instructions following the branch but in the same trace are squashed,

while subsequent traces are not squashed. Within a PE, a simple (non-selective) squash model is preserved, yet at a higher level it appears as if instructions are inserted/removed from the *middle* of the instruction window.

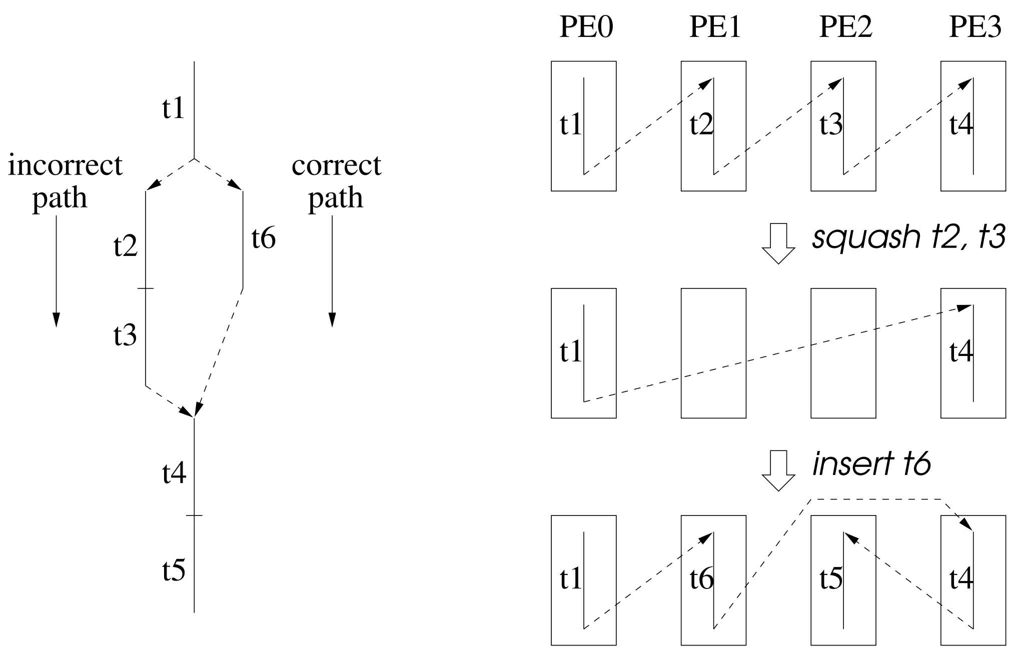
This is called *fine-grain control independence (FGCI)* because the branch and its re-convergent point are close together. Figure 3-13(a) shows an example in which a misprediction in *PE1* affects only control flow within the PE, and inter-trace control flow (links between PEs) is unaffected.

2. *Hierarchical management of resources*. If one or more control dependent paths of a branch are longer than a trace, then recovering from a misprediction involves squashing and inserting an arbitrary number of traces in the middle of the window. To do so, the PEs are managed as a linked-list instead of a fifo. This is no more complex than fifo management, however, because the unit of insertion/removal (a trace) is large and therefore efficient to manage.

   This is called *coarse-grain control independence (CGCI)* because the branch and its re-convergent point are in different traces. Figure 3-13(b) shows an example in which two traces *t2* and *t3* must be removed from the middle of the instruction window and trace *t6* inserted in their place.



(a) Fine-grain control independence (FGCI)



(b) Coarse-grain control independence (CGCI)

Figure 3-13: Flexible window management in a trace processor.

3.4.1.2 Trace selection: ensuring and identifying trace-level re-convergence

Trace-based window management simplifies arbitrary instruction insertion and removal but introduces a new problem. Although control flow eventually re-converges after a branch, *trace-level re-convergence* is not guaranteed. Consider Figure 3-14(a), in which the two control dependent paths of the branch are of different lengths. In Figure 3-14(b), a different set of traces is selected depending on the direction of the branch, even traces after the re-convergent point. I.e. re-convergence is not manifested at the trace-level.

Trace-level re-convergence must be ensured in order to exploit both FGCI and CGCI. This rests with *trace selection*, the algorithm for dividing the dynamic instruction stream into traces. Essentially, trace selection must synchronize the control dependent paths of a branch so that regardless of which path is taken, the same sequence of control independent traces are selected. Traces may be synchronized at the re-convergent point itself, but more generally at any control independent point after the re-convergent point. I develop two different trace selection techniques to address FGCI and CGCI separately.

Small `if-then`, `if-then-else`, and nested `if-then-else` constructs that do not contain loops or function calls are ideally suited to FGCI mechanisms. First, they have fixed-length and relatively short control dependent paths, most of which fit within a trace. Table 6-3 (see Chapter 6) shows that in the worst case, less than 10% of these constructs have a control dependent path longer than 32 instructions. Secondly, they account for a large enough fraction of mispredictions (20% - 60%) to be specially targeted for control independence. Lastly, these regions can be precisely and efficiently detected by hardware because of their directed, acyclic control flow.

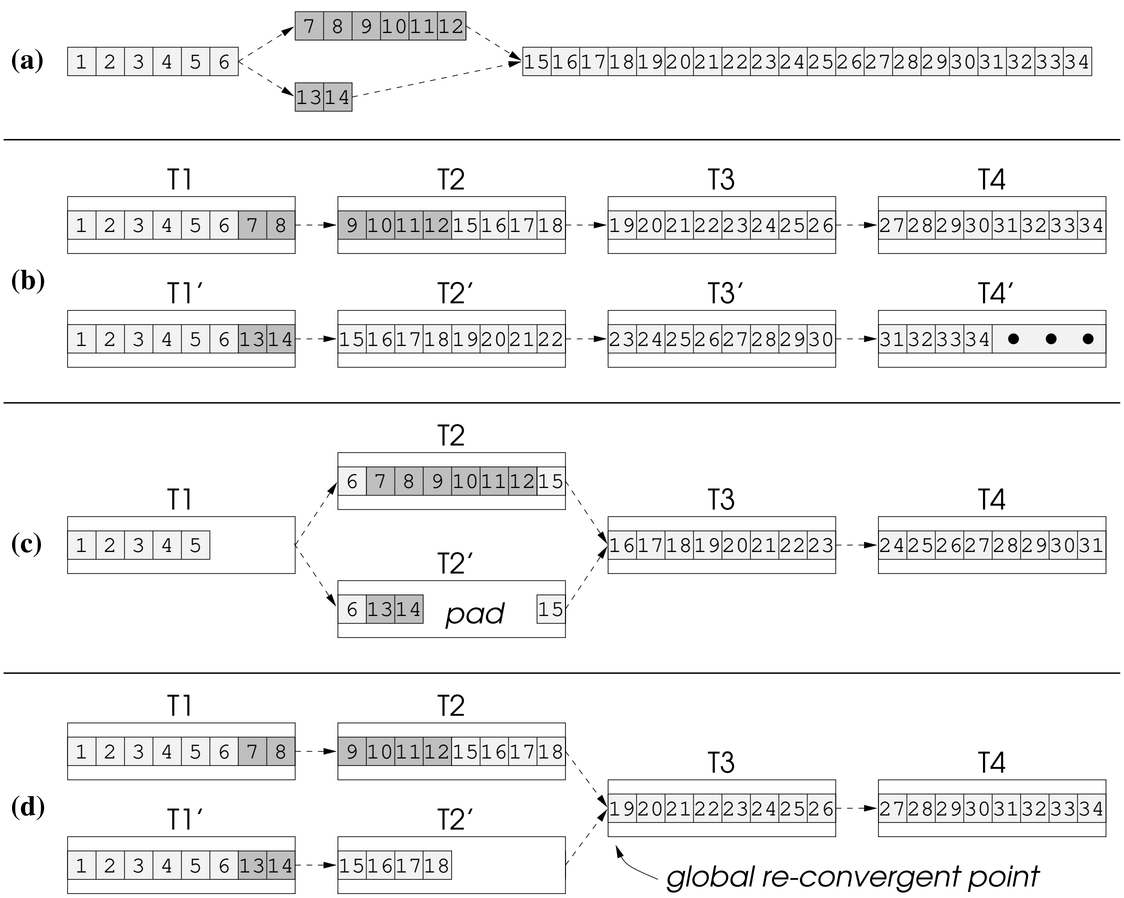


Figure 3-14: The problem of trace-level re-convergence. (a) Instruction-level re-convergence. (b) Default selection does not ensure trace-level re-convergence. (c) FGCI selection and (d) CGCI selection ensure trace-level re-convergence.

I propose a hardware algorithm that detects these forward-branching regions, locates the re-convergent point that closes the region, and computes the length of the longest control dependent path through the region. Trace selection then uses this information to conceptually "pad" any selected path until its length matches the longest path. By equalizing path lengths, trace selection synchronizes control dependent paths at the re-convergent point -- this is shown in Figure 3-14(c). This padding technique enables traces to expand or contract to recover from mispredictions, without affecting the boundaries of subsequent traces, similar to *tasks* in multiscalar processors [100]. Section 3.4.3 describes how forward-branching regions are detected and analyzed, and how trace selection uses the resultant information to expose FGCI.

All other branches are covered by CGCI. In the extreme case, trace selection could search for the precise, i.e. nearest, control independent points for all branches, and then use these points to delineate traces. However, experience with this approach has yielded negative results. There are so many re-convergent points that synchronizing at every one of them creates a large number of small traces, worsening PE utilization, trace cache performance, and trace predictor performance.

Instead, trace selection can exploit a limited number of easily identified, “global” control independent points in the dynamic instruction stream. Loop back-edges, loop exits, and subroutine return points are all examples of global re-convergent points [81,2,82,83]. To ensure trace-level re-convergence for CGCI branches, traces are delineated at chosen global re-convergent points -- as shown in Figure 3-14(d). Then, if a branch is mispredicted, an exposed global re-convergent point nearest the branch is found in the window and assumed to be the first control independent trace. Section 3.4.4 describes CGCI trace selection.

3.4.1.3 Managing data dependences

A mispredicted branch instruction causes not only incorrect control flow, but potentially incorrect data flow as well. After repairing the control flow, control independent instructions must be inspected for incorrect data dependences both through registers and memory, and any incorrect-data dependent instructions selectively reissued.

As we have seen, a primary feature of trace processors is the pervasive use of data speculation. Selective misspeculation recovery was anticipated as an important problem and plays a central role in the trace processor microarchitecture [81]. The selective recovery model as it applies to control independence is reviewed in Section 3.4.2.2.

3.4.2 Trace processor window management

In Section 3.4.1.1, I highlighted the two ways in which trace processors flexibly insert and remove instructions from the middle of the instruction window: FGCI and CGCI. The following two sections provide details regarding control flow and data flow management, respectively.

3.4.2.1 Managing control flow

Sophisticated control flow management is performed by the trace processor frontend since it controls PE allocation (trace dispatch) and deallocation (trace squash). Basic operation of the frontend was described previously in Section 3.1 and depicted in Figure 3-2.

Recall that a trace is not only dispatched to the PE but also placed in the corresponding outstanding trace buffer. The trace buffer monitors branch outcomes as they become available from the PE. If a misprediction is detected, the trace predictor is backed up to that trace, as are the global register rename maps. Also, the trace buffer begins repairing the trace from the point of the branch misprediction, using either the alternate trace prediction or the simple branch predictor to fetch instructions. However, subsequent PEs and their traces are not affected at this point, i.e. they continue processing instructions. When the mispredicted trace has been repaired, the trace-level sequencer is restarted in one of two ways, depending on whether the branch is covered by fine- or coarse-grain control independence.

1. Fine-grain: Control flow recovery for FGCI is very simple because the PE arrangement is unaffected. The frontend merely dispatches the repaired trace from its trace buffer to the affected PE. Within the affected PE, only instructions after the mispredicted branch are squashed.

   At this point, the register rename map reflects correct register dependences up to and including the repaired trace; a *trace re-dispatch sequence*, described in the next subsection, makes a pass through the control independent traces to update their register dependences.

2. Coarse-grain: First, the sequencing hardware locates an exposed global re-convergent point in the window -- the one *after* and generally *nearest* the mispredicted trace. CGCI trace selection (Section 3.4.4) detects and chooses certain global re-convergent points at which to terminate traces; these points are always “exposed” as trace boundaries and, therefore, visible to the sequencing hardware. Note that an exposed global re-conver-

gent point may not exist in the window, in which case CGCI is not exploited. Even if one is located, it may or may not actually be control independent with respect to the mispredicted branch.

Next, the traces between the mispredicted branch and the first (assumed) control independent trace are squashed and their PEs deallocated. The trace predictor fetches the correct control dependent traces and they are allocated to newly freed PEs. Squashing and allocating PEs proceed in parallel, just as dispatch and retirement proceed in parallel. If there are more correct control dependent traces than incorrect ones, then PEs must be reclaimed from the tail (i.e. the most speculative PE).

Finally, the control flow is successfully repaired when re-convergence is detected, i.e. *when the next trace prediction matches the first control independent trace*. Although re-convergence is not guaranteed, if and when it occurs, a trace re-dispatch sequence is performed on the control independent traces to update their register dependences, as described in the next subsection.

With CGCI, the logical or program order of PEs can no longer be inferred from just the head/tail pointers and the *physical* order of PEs. Logically inserting and removing PEs between two arbitrary PEs, i.e. inserting and removing control dependent traces, requires managing the PEs as a linked-list. The linked-list control structure is simply a small table indexed by physical PE number, with each entry containing three fields: logical PE number (order in the list) and pointers to the previous and next PEs. Also, head PE and tail PE pointers are needed as before. The control structure (table plus head/tail pointers) is consulted and possibly updated by the trace-level sequencer when dispatching, retiring, squashing, and re-dispatching traces.

3.4.2.2 Managing data flow

After the sequencing hardware repairs control flow, incorrect-data dependent, control independent instructions must be identified and selectively reissued. The first step is to detect the "source" of a data dependence violation and reissue that instruction. There are two possible sources: 1) a stale physical register name representing an incorrect register dependence, or 2) a load instruction that loaded an incorrect version of a memory location. The second step is to selectively reissue all subsequent data dependent instructions.

Stale physical register names. The frontend initiates a *trace re-dispatch sequence* after control flow is repaired. Control independent traces are re-dispatched in sequential (program) order from the trace buffers to respective PEs. Live-in registers are renamed using the updated maps, and live-out registers do not change their mappings. The source register names of each instruction in the PE-resident trace are checked against those in the re-dispatched trace. Only those instructions with updated register names are reissued.

Incorrect loads. For memory dependences, I leverage the existing mechanism for detecting incorrectly disambiguated loads. Recall from Section 3.3.2, detection of "normal" memory dependence violations is based on loads snooping store addresses and sequence numbers on the cache ports.

This same mechanism works for control independent loads that are incorrectly disambiguated due to mispredicted branches. There are two cases.

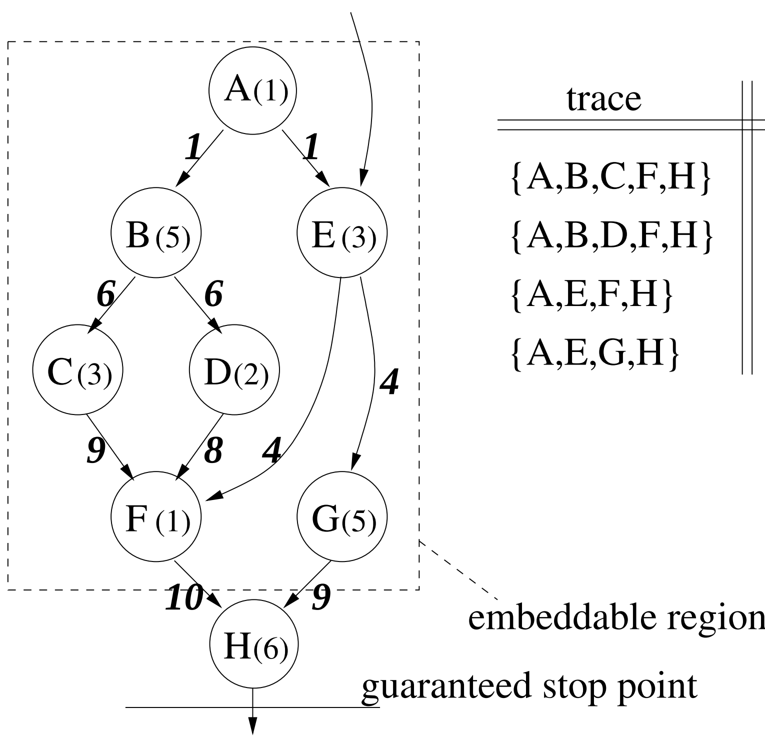
- When a store is removed from the window, i.e. if it is among the incorrect control dependent instructions, it schedules a store undo operation (only if it has performed already).
- A store on the correct control dependent path is brought into the window late, and may appear as a normal disambiguation violation when control independent loads observe the late-performed store.

Sequence number comparisons first require translating physical to logical sequence numbers. In [81], because the PEs are organized in a physical ring, the mapping is fairly direct. Now, due to the arbitrary arrangement of PEs, translation requires consulting the linked-list control structure (each PE maintains a copy). Recall that the linked-list table maintains physical to logical PE translations: this field exists solely for disambiguation support.

Selectively re-issuing dependence chains. After detecting the "source" of a data dependence violation (stale register name or incorrect load), the violating instruction is re-issued. The second step is to selectively re-issue all subsequent data dependent instructions. This happens transparently in the trace processor because instructions remain in the PEs until they retire. Therefore, if an instruction has already issued and it receives one or more additional values, it simply re-issues as many times as is necessary.

3.4.3 Trace selection for FGCI

An example of FGCI trace selection is shown in Figure 3-15. Basic blocks are labeled with a letter *A* through *H*, and block sizes are shown in parentheses. Control flow edges are labeled with the longest path length leading to that edge. The maximum trace length is 16 in the example.



trace	length
{A,B,C,F,H}	16
{A,B,D,F,H}	15
{A,E,F,H}	11
{A,E,G,H}	15

Figure 3-15: Example of an embeddable region.

The branch in block A is a candidate for FGCI because the maximum length of any of its control dependent paths is 10 instructions, well within the maximum trace length. The region enclosed in the dashed box (the branch in block A and its control dependent instructions) is called the *embeddable region*, and the *dynamic region size* of this region is its maximum path length, 10.

During trace construction, if a branch with an embeddable region is encountered, the accrued trace length is incremented by the branch's dynamic region size, *irrespective of which control dependent path is actually selected*. The result will be one of four traces as shown in the table of Figure 3-15. First, not all traces are 16 instructions long, only the trace which actually embeds the longest control dependent path. Second, all traces end at the same instruction, namely the last instruction in basic block H. This of course achieves the desired effect: if the trace predictor predicts one trace and it turns out to be incorrect, it can be replaced with one of the alternate traces without changing the sequence of subsequent control independent traces. Third, any of three branch mispredictions is covered by this region -- the branches in basic blocks *A*, *B*, and *E*.

Exposing FGCI first requires finding branches with embeddable regions (Section 3.4.3.1). A *FGCI-algorithm* is applied to each newly-encountered branch to check if it has an embeddable region. If it does, the goal of the algorithm is to determine 1) the re-convergent PC that closes the region and 2) the dynamic region size. The latter amounts to computing the longest path through a topologically sorted DAG [14] in hardware. This gathered information is cached so that it does not have to be re-computed each time a branch is encountered. Then, trace selection can use the cached information to conceptually pad traces (Section 3.4.3.2).

3.4.3.1 FGCI-algorithm

The underlying idea behind the FGCI-algorithm is to serially scan a static block of instructions following a forward conditional branch. Only a single pass is required and the static block is generally no larger than the maximum trace length, and often much smaller. While scanning, the algorithm simultaneously 1) maintains path lengths and 2) searches for the re-convergent point.

Conceptually, each instruction is modeled as a node having one or more incoming control flow edges, each edge having a value equal to the maximum path length leading to the edge. The algorithm assigns a value to the node equal to the maximum value of incoming edges plus one for the current instruction. In this way, the longest control dependent path lengths are propagated from incoming edges to outgoing edges.

The key to the algorithm, therefore, is determining all incoming edges to an instruction. The implicit edge between two sequential instructions is readily apparent. The edges between forward branches/jumps and their targets require explicit storage. When a forward branch/jump is encountered, its taken target PC is written into a small associative array, called *Targets*. This records the edge for use later, when the target instruction is eventually reached. Of course, the edge must be labeled with a path length, so in addition to writing the target PC, the branch/jump also writes its own computed length into the array. Thus, an edge in *Targets* is a tuple {target PC, length}. When the target instruction is eventually reached, its PC is associatively searched in *Targets* to get the path length of the incoming edge.

Note that multiple branches/jumps may have the same target, a case which appears as multiple incoming edges to the target instruction. This is efficiently handled by *Targets*. Only the edge with the longest path length needs to be remembered. A branch/jump does not write the array if an entry for the target already exists and the stored length is greater than that being written.

Given this overview of computing the maximum path length to any instruction, I can explain how the algorithm computes the re-convergent point and dynamic region size. The algorithm maintains three words of state plus the *Targets* array, which has enough entries to support a reasonable number of targets in a region (e.g. 4 to 6). In the notation below, the current instruction being analyzed is denoted as i. Also, the notation *Targets*(*pc*) means that *pc* is associatively searched in *Targets* and the corresponding path length is output.

1. *pc*: This is a local program counter used for serially scanning the code block. In an implementation, a cache line is fetched from the instruction cache and buffered, and the low bits of *pc* are used to index into the buffer.

2. *length*: This is the maximum path length from the first instruction in the region to the current instruction (scan point). The current instruction modifies *length*, and the new value is a simple function of path lengths from incoming edges.

$$length_i = 1 + \mathrm{MAX}(length_{i\text{-}1}, Targets(pc_i))$$

   Only two edges have to be examined: the edge from the preceding instruction ($length_{i-1}$) and the edge that represents all prior branches/jumps to instruction i ($Targets(pc_i)$). The former edge may not exist if the preceding instruction is an unconditional jump, and the latter edge may not exist if there are no prior branches/jumps to instruction i. If neither exists, then the instruction is unreachable from within the region and therefore does not contribute to the dynamic region size. The *pc* is updated to skip over unreachable instructions.

3. *Targets*: If instruction *i* is a branch/jump with target $target_i$, then *Targets* is updated as follows.

$$\text{if } ((target_i \text{ does not exist in } Targets) \parallel (length_i > Targets(target_i)))$$
$$Targets(target_i) = length_i$$

4. *max_pc*: This is the most distant forward branch/jump target yet seen. If the current instruction is a branch or jump, *max_pc* is overwritten with the target address if it is greater than *max_pc*.

The re-convergent point is reached when *pc* equals *max_pc*. In this case, the branch enclosing the region is a candidate for FGCI. Three pieces of information are gathered for the branch: 1) a flag which indicates this branch has an embeddable region, 2) the re-convergent PC -- simply the value of *max_pc*, and 3) the dynamic region size -- if *i* is the re-convergent instruction, the dynamic region size is equal to $\text{MAX}(length_{i-1}, Targets(pc_i))$.

There are several other terminating conditions which indicate the branch is not a candidate for FGCI. If the value of *length* exceeds the maximum trace length before re-convergence, or if a backward branch, function call, or indirect branch are encountered, scanning is terminated.

The information computed by the FGCI-algorithm is written into a cache called the *branch information table* (BIT). All forward conditional branches allocate entries in the

BIT, whether they have an embeddable region or not, because trace selection needs to know this determination. For a 16K-entry BIT and a trace length of 32 instructions, a BIT entry is 4 bytes long: a tag (16 bits), a flag indicating embeddable or not (1 bit), the dynamic region size (5 bits), and the re-convergent point in the form of an offset from the start of the region (10 bits is reasonable).

In Figure 3-16, the FGCI-algorithm is applied cycle-by-cycle to the example subgraph of Figure 3-15. The diagram shows basic blocks *A* through *G* as they are statically laid out in memory; solid and dashed edges indicate conditional branch targets and direct jump targets, respectively. The four pieces of state are shown varying in time from left to right (each column corresponds to a cycle and a single instruction). If analysis is performed at the rate of 1 instruction/cycle, the latency in this case is at least 21 cycles to compute the re-convergent PC (6e) and dynamic region size (10).

The FGCI-algorithm has several characteristics amenable to hardware implementation. First, it performs a single pass which yields a simple controller and simple state maintenance. Second, limiting analysis to 1 instruction/cycle helps manage complexity as well as reduce bandwidth to the instruction cache. The design is relatively non-intrusive to the cache port -- in the above example, 2 or 3 cache line fetches are initiated over 21 cycles, for a line size of 16 words.

blk	A	E			G					B					C			D		F	H...
pc	0x5a	5b	5c	5d	5e	5f	60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e
length	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9	7	8	10	-
Targets	**63,1**	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1	63,1
	-	-	-	**6d,4**	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	6d,4	**6d,9**	6d,9	6d,9	6d,9	6d,9
	-	-	-	-	-	-	-	-	**6e,9**	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9	6e,9
	-	-	-	-	-	-	-	-	-	-	-	-	-	**6b,6**	6b,6	6b,6	6b,6	6b,6	6b,6	6b,6	6b,6
max_pc	**63**	63	63	**6d**	6d	6d	6d	6d	**6e**	6e	6e	6e	6e	6e	6e	6e	6e	6e	6e	6e	6e

time →

Figure 3-16: FGCI-algorithm applied to subgraph in Figure 3-15.

3.4.3.2 FGCI trace selection

When a forward conditional branch is encountered during trace selection, the branch PC is used to access FGCI information from the BIT. If the information does not exist, a BIT miss handler initiates the FGCI-algorithm and trace construction stalls until the handler completes.

If the BIT indicates the branch is a candidate for FGCI, and the current trace length plus the dynamic region size of the branch does not exceed the maximum trace length constraint, then the following actions are performed.

1. The cumulative trace length is incremented by the dynamic region size.
2. Incrementing of the cumulative trace length is halted while a given path through the embeddable region (as dictated by branch prediction) is added to the trace.
3. Incrementing the cumulative trace length resumes when the re-convergent point (from the BIT) is reached.

Managing the cumulative trace length in this way guarantees paths shorter than the longest path through the embeddable region are effectively "padded" to the longest path length.

If the current trace length plus the dynamic region size of the branch exceeds the maximum trace length constraint, then the current trace is terminated before the branch. Deferring the branch to the next trace ensures all potential FGCI is exposed.

3.4.4 Trace selection and heuristics for CGCI

Trace selection and the trace processor frontend coordinate to exploit CGCI. Trace selection delineates traces at key global re-convergent points. When a misprediction is detected, the frontend chooses one of the points, if there are any in the window, to serve as the trace-level re-convergent point for recovery.

3.4.4.1 Trace selection: exposing global re-convergent points

In this thesis, I consider only two types of global re-convergent points. These are the targets of return instructions and the not-taken targets of backward branches, shown with black dots in Figure 3-17(a) and Figure 3-17(b), respectively.

The default trace selection algorithm terminates traces at the maximum trace length or at any indirect branch instruction; indirect branches include jump indirect, call indirect, and return instructions. Therefore, default trace selection already ensures trace-level re-convergence at the exits of functions, i.e. return targets.

An additional CGCI trace selection constraint, called *ntb*, terminates traces at predicted not-taken backward branches. This ensures trace-level re-convergence at the exits of loops.

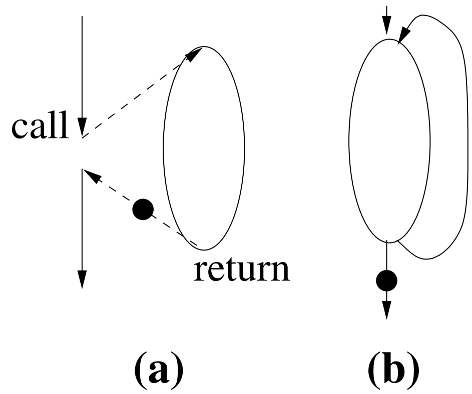


Figure 3-17: Trace selection for exposing global re-convergent points.

3.4.4.2 CGCI heuristics: choosing a global re-convergent point for recovery

Trace selection only ensures trace-level re-convergence at easily identified, global re-convergent points. When a branch misprediction is detected, one from possibly many such points in the window must be chosen as the trace-level re-convergent point used for recovery actions. Only two CGCI heuristics are considered and are described below.

- *RET*: The frontend locates the nearest trace that ends in a return instruction. The immediately subsequent trace is assumed to be the first control independent trace.
- *MLB-RET*: If the mispredicted branch is a backward branch, I assume it is a loop branch as depicted in Figure 3-17(b). Based on this assumption, the frontend locates the nearest trace whose starting PC matches the not-taken target of the branch; it is likely the correct re-convergent point. This heuristic, Mispredicted Loop Branch (*MLB*), is always considered first. However, if the mispredicted branch is not a backward branch, then the *RET* heuristic is applied.

The *RET* heuristic is designed to cover arbitrary mispredictions within a function since control flow re-converges at the function exit. However, due to nested functions, there may be any number of other return instructions before the intended function exit. Choosing the nearest return instruction, therefore, is only a guess. Often, the result is better than intended -- when the chosen return is closer than the function exit, but still control independent with respect to the misprediction. Other times, however, the chosen return is on the incorrect control dependent path.

The *MLB* heuristic (of *MLB-RET*) is designed to specifically and accurately cover mispredicted loop branches, a substantial source of branch mispredictions (refer to Table 6-3). Loops with small bodies and a small but unpredictable number of iterations fall in this category, and there are often many control independent traces after the mispredicted loop branch.

The *RET* heuristic requires only default trace selection, whereas *MLB-RET* requires, in addition, the *ntb* selection constraint to expose loop exits.

Chapter 4

Experimental Method

4.1 Simulator

The experiments in this thesis are based on a detailed and fully execution-driven trace processor simulator co-authored by myself and Quinn Jacobson. The simulator was developed using the Simplescalar toolkit [11]. A high-level block diagram of the simulator infrastructure is shown in Figure 4-1. There are four major components in the simulator infrastructure: simplescalar binary, functional simulator, timing simulator, and debug buffer.

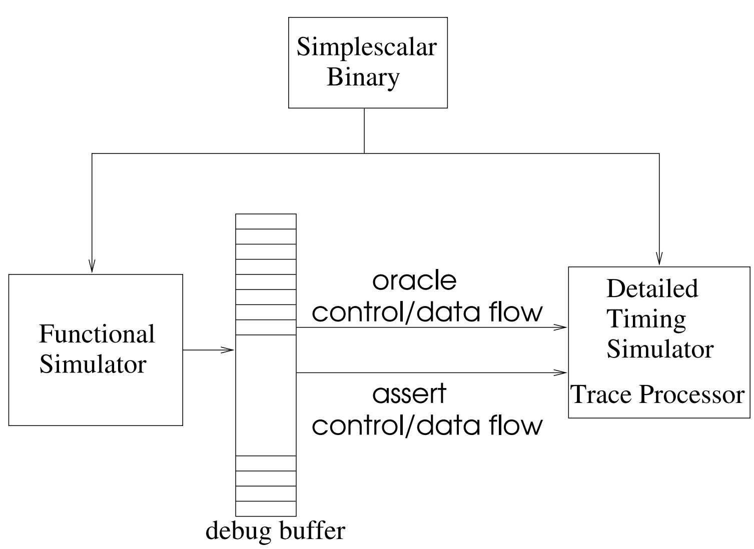


Figure 4-1: Simulator infrastructure.

Benchmarks are compiled to create a *simplescalar binary*. The compiler is gcc-based (gnu C compiler) and targets the simplescalar instruction set architecture (ISA). The ISA resembles the MIPS ISA, however, one major difference is that delayed branches are eliminated from the architecture.

There are two simulators that run in parallel, a *functional simulator* and a *timing simulator*. **The two simulators are completely independent**. They independently fetch and execute instructions from the binary and manage their own copies of the program state. The functional simulator runs ahead of the timing simulator and pushes control and data flow information for each executed instruction onto the *debug buffer* (whose function is described later).

The timing simulator is the bulk of the infrastructure. It faithfully models the trace processor microarchitecture as described in Chapter 3. The timing simulator is fully execution-driven in the following sense:

- Instructions are fetched and executed speculatively based on trace and branch prediction. There is no explicit foreknowledge of whether a prediction is correct or incorrect. Therefore, instruction fetching and execution proceeds along both correctly and incorrectly speculated paths.
- Instructions are executed; they consume and produce values, and these values are communicated among instructions, register state, and memory state.

This approach is essential for accurately modeling data speculation and control independence. Instructions may execute with incorrect speculative values and re-issue any number of times upon receiving new values; loads may pollute the data cache (or inadvertently prefetch) with wrong addresses; extra bandwidth demand is observed on result buses, etc.

The timing simulator is quite complex and some level of verification is essential. The functional simulator is known to be correct because 1) it is a very simple architectural simulator, not a timing simulator (its core is *sim-fast* of the Simplescalar toolkit), and 2) it only needs to be verified once for each benchmark. Limited verification is provided by the debug buffer, the only interface between the functional and timing simulators. Before retiring each instruction, the timing simulator pops what should be the corresponding, known-correct instruction information from the debug buffer. The PC and operands of the retired instruction are compared to the known-correct debug buffer state.

In addition to verification, the debug buffer provides another indispensable capability. Perfect data and control speculation is possible by consulting the debug buffer for oracle predictions (the reason the functional simulator must run ahead of the timing simulator). The two debug buffer interfaces are labeled in the diagram as “assert control/data flow” and “oracle control/data flow”.

4.2 Configuring the trace processor

Trace processor parameters are shown in Table 4-1. The table is organized hierarchically, e.g. the trace processor frontend has several components and each component has several parameters. Ultimately, individual parameters are described and, in most cases, default configurations are given.

Several key parameters are varied within and among chapters. These are indicated with shaded entries in Table 4-1, and I discuss them below. Other parameters may depend on the shaded entries in a deterministic way, e.g. the number of local result buses is always equal to PE issue width; such parameters are not discussed further since their relationship to other parameters is fixed.

- *Trace cache*.

  A wide range of trace cache configurations are evaluated in Section 5.1. In particular, trace cache size, associativity, and indexing (PC-only indexing versus path associative indexing) are varied.

  All other chapters use the default size, associativity, and indexing indicated in Table 4-1. Trace line size varies, however, in which case the number of trace cache lines is adjusted to maintain a constant trace cache size (instruction storage only). The trace line size is either 16 or 32 instructions, corresponding to 2K or 1K trace cache lines, respectively.

- *Trace construction*.

  All experiments except those in Section 5.1 use the default trace construction parameters.

  Section 5.1 changes only one parameter, the fetch bandwidth from the instruction cache when the second-level branch predictor drives instruction fetching (see Section 3.1.2). This parameter is the last bullet under “trace processor frontend -> trace construction ->

trace construction bandwidth". A slightly more aggressive instruction cache design is used. When the second-level branch predictor is used to fetch from the instruction cache, any number of *sequential* basic blocks up to a full cache line may be fetched in a single cycle, instead of just one basic block per cycle. This rule is identical to the third bullet under "trace processor frontend -> trace construction -> trace construction bandwidth", and requires either an interleaved BTB [13,79] or a merged BTB/instruction cache design [84] to allow fast, parallel prediction of any number of not-taken branches.

- *Trace selection*.

  Trace selection varies in only two ways. Firstly, the maximum trace length is either 16 or 32 instructions. The more fundamental variations involve exposing control independence, i.e. FGCI and CGCI trace selection, used only in Chapter 6.

- *Trace processor dimensions*.

  The trace processor "dimensions" are trace length (PE window size), number of PEs, and PE issue bandwidth. These parameters are frequently varied, particularly in Chapter 5, which evaluates the distributed instruction window in substantial detail.

- *Global register file*.

  In all evaluation chapters, the size of the global register file is guaranteed not to limit performance, i.e. there is an unlimited number of physical registers. In Section 5.4.3, however, I determine the actual number of global physical registers that are required for several key trace processor configurations.

- *Global result buses*.

  In Chapter 5, global result bus bandwidth is unconstrained for many experiments: the number of global result buses is equal to the aggregate issue bandwidth. As explained in the appropriate subsections, unconstrained global result bus bandwidth is necessary to isolate other performance factors relevant to a distributed instruction window. One subsection in the chapter is devoted to measuring sensitivity to global result bus bandwidth, however.

  In other chapters, the number of global result buses is chosen to be commensurate with trace processor configuration (dimensions). In Section 5.1, a superscalar processor is used, in which case there is no register hierarchy and the number of result buses equals the issue bandwidth. Refer to Section 4.4 for a description of modeling superscalar processors using the common simulator infrastructure.

  Global result bus latency is also frequently varied in Chapter 5, but elsewhere it is fixed at one cycle.

- *Cache/ARB buses*.

  The number of buses, or ports, to the data cache and ARB subsystem is chosen to be commensurate with the particular trace processor configuration. Table 4-2 shows for each trace processor configuration 1) the number of cache/ARB buses and 2) the number of buses that can be arbitrated by a single PE.

Table 4-1: Trace processor parameters and configuration.

<table>
<tr><td rowspan="5">trace processor frontend</td><td>trace cache</td><td>size = 128kB (instruction storage only)
assoc. = 4-way
trace line = 16 or 32 instructions
repl. = LRU
path associative indexing
written upon constructing a trace (not at retirement)</td></tr>
<tr><td>trace predictor</td><td>Hybrid:
2^{16}-entry path-based predictor; DOLC = {7,3,6,8}
2^{16}-entry simple predictor
Return History Stack
Correlated predictor entry: tag, primary id, alternate id, 2-bit repl. ctr.
Simple predictor entry: trace id, 4-bit repl. ctr.</td></tr>
<tr><td>trace construction</td><td>1 outstanding trace buffer per PE
Trace construction bandwidth:
• All trace constructors share a single datapath, or port, to the instruction cache, branch predictor, and optional BIT.
• The port is non-blocking: if one trace buffer is awaiting an instruction cache miss, another trace buffer can use the port.
• When instruction cache fetching is driven by the trace predictor (primary or alternate trace id), the instruction cache can fetch any number of sequential basic blocks up to a cache line in a single cycle.
• When the second-level branch predictor is required to fetch from the instruction cache, only one basic block up to a cache line can be fetched in a single cycle.</td></tr>
<tr><td>instruction cache</td><td>2-way interleaved (can fetch through cache line boundaries)
size = 64kB
assoc. = 4-way
repl. = LRU
line size = 16 instructions
miss penalty = 12 cycles (1 for miss, 10 to service, 1 for re-lookup)</td></tr>
<tr><td>branch predictor (BTB)</td><td>size = 16K entries
assoc. = direct mapped and tagless
entry = 2-bit predictor plus taken target
BTB miss handling: Hit or miss is not determined in the normal manner since there is no tag. However, branch targets are computed during trace construction and a 1-cycle penalty is incurred if the branch is predicted-taken and the taken target from the BTB does not match the computed taken target.</td></tr>
</table>

Table 4-1: Trace processor parameters and configuration.

trace processor frontend (cont.)	BIT	*Only used for FGCI trace selection.* size = 8K entries assoc. = 4-way BIT miss handling: • Unlimited MSHRs. An MSHR invokes the FGCI-algorithm. • Latency of the FGCI-algorithm is equal to dynamic region size. • Contention for the instruction cache port is not modeled because the FGCI-algorithm requires little instruction cache bandwidth.
	trace selection	Default: stop at maximum trace length (16 or 32) and indirect branches. Indirect branches include jump indirect, call indirect, and return instructions. Other: FGCI and CGCI trace selection
PE management	Up to 2 PEs can be reclaimed (1 trace retirement and 1 trace squash) and 1 PE allocated (1 trace dispatch) per cycle.	
trace processor “dimensions”	trace length (PE window size)	*varies* (16 or 32)
	number of PEs	*varies* (2,4,8,16)
	PE issue width	*varies* (1,2,4)
hierarchical register file	local register file	physical registers = unlimited read ports = 2 times PE issue bandwidth write ports = 1 times PE issue bandwidth
	global register file	physical registers = unlimited read ports = 2 times PE issue bandwidth (due to replication) write ports = # of global result buses
functional units	*n* symmetric, fully-pipelined FUs (for *n*-way issue)	

operand bypasses	local result buses	# buses = PE issue bandwidth
		latency = 0 (i.e., local bypass occurs at end of execution stage)

	global result buses	# buses = *varies*
		latency = *varies* (default: 1 cycle)

pipeline latencies	fetch	1 cycle
	dispatch	1 cycle
	issue	1 cycle
	execution latencies	address generation = 1 cycle
		cache access = 2 cycles (hit)
		integer ALU ops = 1 cycle
		complex ops = MIPS R10000 latencies
selective recovery	load re-issue penalty	1 cycle : models the snoop latency
	value misp. penalty	1 cycle : models the operand validation latency

Table 4-1: Trace processor parameters and configuration.

<table>
<tr><td rowspan="3">data memory subsystem</td><td>data cache</td><td>size = 64kB
assoc. = 4-way
repl. = LRU
line size = 64 bytes
miss penalty = 14 cycles (2 for miss, 10 to service, 2 for re-lookup)
MSHRs = unlimited outstanding misses</td></tr>
<tr><td>ARB</td><td>Unlimited speculative store buffering:
• Unbounded number of addresses managed.
• Unbounded number of versions per address.
Ideal speculative state commit (never stalls retirement).
Store undo bandwidth:
• Modeled for speculative disambiguation.
• Not modeled for selectively squashing control dependent path.</td></tr>
<tr><td>cache/ARB buses (ports)</td><td>varies (commensurate with trace processor dimensions; see Table 4-2)</td></tr>
<tr><td>unified L2 cache</td><td colspan="2">perfect</td></tr>
</table>

Table 4-2: Configuring cache/ARB buses for trace processors.

trace processor configuration			# cache/ARB buses	# buses arbitrated per PE
trace length (PE window size)	# PEs	PE issue bandwidth		
16	2	1	1	1
16	2	2	2	2
16	2	4	2	2
16	4	1	2	1
16	4	2	4	2
16	4	4	4	4
16	8	1	4	1
16	8	2	4	2
16	8	4	4	4
16	16	1	8	1
16	16	2	8	2
16	16	4	8	4
32	2	1	2	1
32	2	2	4	2
32	2	4	4	4
32	4	1	4	1
32	4	2	4	2
32	4	4	4	4
32	8	1	8	1
32	8	2	8	2
32	8	4	8	4
32	16	1	8	1
32	16	2	8	2
32	16	4	8	4

4.3 Value predictor configuration

Live-in prediction is applied and evaluated only in Section 6.1. A context-based value predictor is used. Context-based predictors learn values that follow a particular sequence of previous values [88].

The predictor is organized as a two-level table. The first-level table is indexed by a unique *prediction id*, derived from the trace id. A given trace has multiple prediction ids, one per live-in register in the trace. An entry in the first-level table contains a pattern that is a hashed version of the previous 4 data values of the item being predicted. The pattern from the first-level table is used to look up a 32-bit data prediction in the second-level table. Replacement is guided by a 3-bit saturating counter associated with each entry in the second-level table.

The predictor also assigns a confidence level to predictions [37,51]. Instructions issue with predicted values only if the predictions have a high level of confidence. The confidence mechanism is a 2-bit saturating counter stored with each pattern in the first-level table.

The table sizes used in Section 6.1 are very large in order to explore the potential of such an approach: 2^{18} entries in the first-level, 2^{20} entries in the second-level. Accuracy of context-based value prediction is affected by timing of updates, which I accurately model. A detailed treatment of the value predictor can be found in [88].

4.4 Modeling superscalar processors

In Chapter 5, I compare superscalar and trace processors. The timing simulator is designed to seamlessly simulate either superscalar or trace processors. A necessary consequence is that the superscalar processor shares many of the same beneficial qualities of the trace processor -- in particular, the hierarchical frontend (trace cache and trace predictor) and advanced data flow management (memory dependence prediction with selective recovery).

The only differences between superscalar and trace processor modes are the window organization (centralized versus distributed) and the register forwarding mechanism (single level versus hierarchical).

1. Centralized instruction window.

   In trace processor mode, the number of outstanding traces in the processor may not exceed the fixed number of PEs. In superscalar mode, the PE simulator construct is still used to form an instruction window -- this maximally leverages the underlying simulator infrastructure. The appearance of a centralized instruction window is achieved by providing the superscalar processor with a virtually unlimited number of PEs and outstanding trace buffers to match. Therefore, the number of PEs does not constrain the size of the instruction window that can be formed, which is essential since trace selection often creates traces shorter than the PE window size. Instead, a superscalar window size parameter constrains the total number of *instructions* in the processor and the number of *traces* is irrelevant.

Finally, the per-PE issue bandwidth constraint is substituted with a single, shared issue bandwidth constraint. There is no limit to the number of instructions that may issue from one trace or another, other than the superscalar issue bandwidth.

2. Non-hierarchical register forwarding.

   The superscalar mode must appear to have a single level of result buses for operand bypassing. This is achieved by setting the number of local result buses and global result buses equal to the superscalar issue bandwidth, and when an instruction completes, it writes its result to both a local and global result bus (logically one result bus). Also, the local and global result bypass latencies are both set equal to 0.

As I did for trace processor configurations, Table 4-3 shows the number of cache/ARB buses for superscalar processor configurations.

Table 4-3: Configuring cache/ARB buses for superscalar processors.

superscalar processor configuration		# cache/ARB buses
window size	issue bandwidth	
16	1	1
16	2	1
16	4	2
32	1	1
32	2	1
32	4	2
32	8	4
64	2	1
64	4	2
64	8	4
64	16	4
128	4	2
128	8	4
128	16	4
256	4	4
256	8	8
256	16	8

4.5 Benchmarks and trace characterization

The SPEC95 integer benchmarks are used for evaluation, each of which is simulated to completion. Benchmarks, input datasets, and instruction counts are shown in Table 4-4.

Table 4-4: Benchmarks.

benchmark	input dataset			dynamic instr. count
	spec dir.	**command line arguments**	**other relevant files**	
compress[a]	-	400000 e 2231	-	104 million
gcc	ref	-O3 genrecog.i -o genrecog.s	-	117 million
go	-	9 9	-	133 million
jpeg[b]	train	vigo.ppm	-	166 million
li	test	test.lsp (queens 7)	-	202 million
m88ksim	train	-c < ctl.in	dcrand.lit	119 million
perl	train	scrabble.pl < scrabble.in[c]	dictionary	108 million
vortex	train	vortex.in[d]	persons.250, lendian.wnv, lendian.rnv	101 million

a. Benchmark source code modified to make only a single compress/decompress pass.

b. Benchmark source code modified to hardwire in the following (otherwise verbose) arguments: -image_file vigo.ppm -compression.quality 15 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp

c. Two more words were added to “scrabble.in”. The file now contains {zed, veil, vanity, abdome, evilds}.

d. The following entries in “vortex.in” were modified: part_count=100, inner_loop=1, lookups=20, deletes=20, stuff_parts=10, pct_newparts=10, pct_lookups=10, pct_deletes=10, pct_stuffparts=10.

The traces produced by default trace selection are characterized for each benchmark in Table 4-5 and Table 4-6, for length-16 and length-32 traces, respectively. The minimum, maximum, average, and standard deviation are given for each trace characteristic.

The average trace length for maximum-length-16 traces varies from 12.4 (*li*) to 15.8 (*jpeg*) instructions. The average trace length for maximum-length-32 traces varies from

19.7 (*li*) to 31.1 (*jpeg*) instructions (a 60% to 100% increase over maximum-length-16 traces).

On average, length-16 traces have about 5 live-in registers, 5 or 6 live-out registers, and 4 or 5 local registers. Length-32 traces have on average about 5 or 6 live-in registers, 8 live-out registers, and 10 local registers. In going from length-16 to length-32 traces, the number of local registers increases more rapidly than either live-in or live-out registers. For all benchmarks, the number of local registers more than doubles (125% increase on average). In contrast, live-in registers grow by about 10% and live-out registers by about 40%. In any case, the number of local registers is significant: the amount of write traffic to the global file and global operand bypasses is reduced by 45% and 57% on average, for length-16 and length-32 traces, respectively.

Length-16 traces contain an average of 1.8 conditional branches and 2.4 total control transfer instructions. Length-32 traces contain an average of 3.1 conditional branches and 4.2 total control transfer instructions.

The last two trace characteristics shown in Table 4-5 and Table 4-6 are trace cache miss and misprediction “penalties”. They depend on the trace construction model and timings, and must be evaluated in the context of a trace processor organization. I used a trace processor with 8 PEs, 2-way issue per PE, 4 global result buses, and a global operand bypass latency of 1 cycle. I place quotes around the word “penalty” because the latency for constructing a trace is not a fixed number (whereas a penalty is usually interpreted as a fixed latency, e.g. L1 data cache miss penalty) -- instead, trace construction latency depends on the number of branches and instructions in the trace, the availability of BTB and instruction cache information, the availability of an alternate trace id in the case of mispredictions, and generally complex timings of the trace construction mechanism which is modeled in great detail.

Each “penalty” is measured as the number of cycles required to construct (trace cache miss) or re-construct (trace misprediction) a trace *during the times that the trace constructor has access to the shared instruction cache port*; if trace construction is stalled because another trace constructor is using the cache port, the stall time is not included in the construction time of the first trace. Therefore, trace construction time includes cycles required to fetch instructions from the instruction cache and predict branches using the BTB, and both instruction cache and BTB miss latencies are included (a BTB miss for a predicted-taken branch instruction incurs a one cycle penalty for decoding the branch and computing the target). For reference, the trace construction mechanism is described in detail in Section 3.1.2, and specific configuration information is given in Table 4-1.

When interpreting the trace cache miss and misprediction “penalties”, one should keep in mind that the latencies may partially overlap. For example, if there are multiple instruction cache ports, then multiple trace constructors may operate in parallel in the event of back-to-back trace cache misses (Section 3.1.2). Even in the default configuration, which allows only one trace constructor to use the single cache port *in a given cycle*, if a trace constructor encounters an instruction cache or BTB miss, another trace constructor may take over the port in the interim.

For length-16 traces, average trace cache miss latencies range from 2 to 5 cycles. Re-constructing a mispredicted trace incurs less latency, presumably because 1) fewer instructions must be fetched to construct only the tail of a trace and 2) often an alternate trace id is available and it hits in the trace cache (Section 3.1.2) -- on average, between 1 and 3 cycles are required to re-construct the trace. Of course, trace mispredictions are much costlier than trace cache misses in terms of overall performance, because mispredictions are detected many cycles after fetching traces, whereas misses are detected at fetch time.

As one would expect, trace construction latencies increase for length-32 traces. Average trace cache miss latencies range from 3.5 to 9 cycles, and re-constructing a mispredicted trace requires 2 to 4 cycles on average.

Table 4-5: Trace characteristics for length-16 traces.

statistic		comp	gcc	go	jpeg	li	m88k	perl	vortex
trace length (inst)	min/max	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16
	avg.	**14.5**	**13.9**	**14.8**	**15.8**	**12.4**	**13.0**	**13.0**	**14.4**
	std. dev.	2.8	4.1	3.4	1.5	5.1	5.2	5.0	3.3
live-ins	min/max	0/14	0/17	0/19	0/17	0/15	0/20	0/17	0/17
	avg.	**5.2**	**4.4**	**5.0**	**6.8**	**4.1**	**3.9**	**4.0**	**5.5**
	std. dev.	1.8	2.7	2.3	3.1	2.3	1.7	2.5	2.7
live-outs	min/max	0/13	0/14	0/16	0/14	0/16	0/21	0/16	0/15
	avg.	**6.2**	**5.7**	**5.8**	**6.4**	**5.1**	**4.3**	**5.2**	**6.3**
	std. dev.	2.4	2.7	2.4	2.6	2.2	2.5	2.7	2.7
locals	min/max	0/12	0/13	0/13	0/14	0/11	0/14	0/11	0/12
	avg.	**5.6**	**3.7**	**5.9**	**7.1**	**2.5**	**4.9**	**3.4**	**2.7**
	std. dev.	2.4	3.0	3.0	3.1	2.2	3.2	2.7	3.0
loads	min/max	0/12	0/14	0/13	0/11	0/16	0/13	0/16	0/13
	avg.	**2.1**	**3.6**	**3.1**	**2.9**	**3.7**	**2.5**	**3.7**	**4.2**
	std. dev.	1.3	2.7	1.9	2.4	2.6	2.2	2.7	2.6
stores	min/max	0/16	0/16	0/16	0/13	0/14	0/13	0/16	0/16
	avg.	**0.9**	**2.0**	**1.0**	**1.2**	**2.2**	**1.6**	**2.4**	**3.5**
	std. dev.	1.2	2.9	1.9	1.7	2.5	2.0	2.5	3.2
cond. branches	min/max	0/8	0/11	0/10	0/8	0/8	0/11	0/9	0/9
	avg.	**2.1**	**2.1**	**1.8**	**1.0**	**1.9**	**1.8**	**1.7**	**1.7**
	std. dev.	1.5	1.8	1.4	1.4	1.9	1.7	1.5	1.5
control inst.	min/max	0/8	0/11	0/10	0/9	0/9	0/11	0/9	0/9
	avg.	**2.9**	**2.8**	**2.3**	**1.3**	**2.9**	**2.5**	**2.5**	**2.3**
	std. dev.	1.4	1.7	1.3	1.8	1.8	1.6	1.5	1.4
trace cache miss penalty[a]	min/max	2/12	1/50+	1/50+	1/50+	1/12	1/29	1/42	1/50+
	avg.	**5.3**	**5.1**	**4.4**	**3.9**	**2.0**	**2.8**	**2.0**	**5.3**
	std. dev.	4.7	6.8	5.6	5.1	0.8	1.6	1.5	7.0
trace mispredict penalty[a]	min/max	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+
	avg.	**1.7**	**2.5**	**1.9**	**1.4**	**1.5**	**1.7**	**1.6**	**2.6**
	std. dev.	1.3	3.3	2.0	1.3	1.2	1.7	1.9	4.1

a. See text for comments on how "penalties" are characterized.

Table 4-6: Trace characteristics for length-32 traces.

statistic		comp	gcc	go	jpeg	li	m88k	perl	vortex
trace length (inst)	min/max	1/32	1/32	1/32	1/32	1/32	1/32	1/32	1/32
	avg.	**24.9**	**24.0**	**27.2**	**31.1**	**19.7**	**23.8**	**21.2**	**25.6**
	std. dev.	9.6	10.0	8.7	4.1	11.4	9.6	11.6	8.6
live-ins	min/max	1/14	0/19	0/21	1/18	0/24	0/24	0/26	0/20
	avg.	**5.1**	**5.0**	**6.1**	**8.8**	**4.4**	**4.3**	**4.3**	**6.3**
	std. dev.	1.7	3.1	2.8	4.0	2.5	1.7	2.8	2.9
live-outs	min/max	0/17	0/21	0/21	0/20	0/25	0/21	0/20	0/18
	avg.	**7.6**	**8.0**	**8.1**	**9.3**	**6.4**	**6.0**	**6.9**	**9.3**
	std. dev.	3.1	3.6	3.4	3.5	2.8	2.9	3.5	3.8
locals	min/max	0/30	0/27	0/28	0/27	0/23	0/31	0/24	0/25
	avg.	**12.6**	**8.0**	**13.2**	**17.2**	**5.6**	**10.7**	**7.0**	**6.5**
	std. dev.	7.4	6.3	6.6	5.3	5.0	6.4	5.9	4.6
loads	min/max	0/15	0/20	0/20	0/20	0/24	0/17	0/21	0/21
	avg.	**3.6**	**6.3**	**5.6**	**5.8**	**5.9**	**4.6**	**6.0**	**7.5**
	std. dev.	2.3	3.8	2.8	3.9	4.0	3.0	3.7	4.2
stores	min/max	0/28	0/24	0/24	0/24	0/25	0/24	0/26	0/29
	avg.	**1.6**	**3.4**	**1.9**	**2.4**	**3.5**	**2.9**	**3.8**	**6.2**
	std. dev.	2.0	4.5	3.0	2.9	4.2	3.4	3.8	5.8
cond. branches	min/max	0/15	0/22	0/15	0/13	0/16	0/18	0/15	0/16
	avg.	**3.6**	**3.7**	**3.3**	**2.0**	**3.1**	**3.3**	**2.8**	**3.0**
	std. dev.	2.7	3.1	2.4	2.6	3.2	2.5	2.5	2.0
control inst.	min/max	1/15	0/22	0/15	0/14	1/16	0/18	0/16	0/16
	avg.	**5.0**	**4.8**	**4.1**	**2.5**	**4.7**	**4.6**	**4.1**	**4.1**
	std. dev.	2.5	2.9	2.2	3.4	3.3	2.5	2.7	2.0
trace cache miss penalty[a]	min/max	1/15	1/50+	1/50+	1/50+	1/12	1/38	1/50+	1/50+
	avg.	**9.2**	**6.8**	**6.2**	**4.2**	**4.4**	**3.9**	**3.5**	**6.3**
	std. dev.	3.2	9.1	8.0	4.8	1.9	2.0	2.1	8.6
trace mispredict penalty[a]	min/max	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+	1/50+
	avg.	**2.8**	**4.1**	**3.2**	**2.8**	**2.4**	**2.9**	**2.2**	**3.7**
	std. dev.	2.4	5.1	3.4	2.5	2.7	2.9	3.1	5.7

a. See text for comments on how “penalties” are characterized.

Chapter 5

Evaluation of Hierarchy

This chapter presents four studies of the trace processor's hierarchical organization. The first focuses on the trace cache fetch mechanism and the other three focus on the hierarchical instruction issue and register communication mechanisms, as outlined below.

- *Trace cache*.

  The trace processor frontend provides high instruction fetch bandwidth with low latency by predicting and caching sequences of multiple, possibly noncontiguous basic blocks. Section 5.1 evaluates the performance benefits of the trace cache over otherwise equally sophisticated, contiguous instruction fetch mechanisms.

- *Comparative study of superscalar and trace processors*.

  A comparison between conventional superscalar processors and trace processors is presented in Section 5.2. I demonstrate that distributed instruction windows sacrifice some amount of ILP performance for reduced circuit complexity and potentially shorter cycle times. This tradeoff is investigated in depth. Specific conclusions may be drawn directly from the study. But more importantly, the results are presented in such a way that the reader may provide his or her own technology context to arrive at different (or similar) conclusions.

- *Three trace processor dimensions.*

  A non-hierarchical instruction window is typically characterized by two "dimensions", window size and issue bandwidth. In a distributed instruction window, these aggregate dimensions are broken down further into three new and relatively unexplored dimensions: PE window size, number of PEs, and issue bandwidth per PE. Seemingly equivalent trace processors may be derived from different combinations of PE window size, number of PEs, and PE issue width, with unknown performance implications. The decision to increase a particular dimension instead of or at the expense of another dimension also has unknown ramifications. Therefore, the chapter's third study is devoted to understanding the three trace processor dimensions.

- *Hierarchical register model.*

  Distribution is based on the premise of locality, i.e., that a fraction of the overall computation can be performed primarily locally and quickly within a processing element. This premise is tested in the fourth and final study of this chapter, which evaluates the locality-motivated, hierarchical register storage and forwarding mechanisms of trace processors.

As indicated in the methodology chapter, I focus on the integer programs of the SPEC95 benchmark suite due to their irregular, relatively less-predictable control flow and irregular instruction-level parallelism. I chose to omit full results for floating point benchmarks because they are more regular and amenable to a range of ILP techniques (e.g., vector architectures). Nevertheless, the primary experiments (those in Section 5.2.3) are repeated for SPEC95 floating point benchmarks in Section 5.5.

5.1 Hierarchical instruction supply

The trace cache is evaluated in the context of a superscalar processor so that artifacts of a hierarchical instruction window do not obscure the study. For example, several fetch models are defined for comparative studies; these fetch models produce what are effectively short "traces" and mapping a short instruction sequence to a PE is inefficient. It is more appropriate to use an execution model that is not closely tied to a particular fetch model. Therefore, a 16-way superscalar processor with a 256 instruction window is used in all experiments. Other aspects of the processor are configured as described in Section 4.2 and Section 4.4.

This section is organized into three subsections. I begin by defining fetch models used to evaluate the trace cache. Overall performance of the frontend is a combination of trace cache and trace predictor performance; a subsection is devoted to this topic and a technique for isolating the performance of either component is developed. The final subsection presents results.

5.1.1 Fetch models

To evaluate the performance of the trace cache, I compare it to several more constrained fetch models. I first determine the performance advantage of fetching multiple *contiguous* basic blocks per cycle over conventional single block fetching. Then, the benefit of fetching multiple *noncontiguous* basic blocks is isolated.

In all models a next trace predictor is used for control prediction, for two reasons. First, next trace prediction is highly accurate, and whether predicting one or many branches at a time, it is comparable to or better than some of the best single branch predictors in the literature. Second, it is desirable to have a common underlying predictor for all fetch models so I can separate performance due to fetch bandwidth from that due to branch prediction (more on this in Section 5.1.2).

What differentiates the following models is the trace selection algorithm.

- SEQ.1 (*"sequential, 1 block"*): A "trace" is a single basic block up to 16 instructions in length.
- SEQ.n (*"sequential, n blocks"*): A "trace" may contain any number of sequential basic blocks up to the 16 instruction limit.
- TC (*"trace cache"*): A trace may contain any number of conditional branches, both taken and not-taken, up to 16 instructions or the first indirect branch.

The SEQ.1 and SEQ.n models do not use a trace cache because an interleaved instruction cache is capable of supplying a "trace" in a single cycle [29] -- a consequence of the sequential selection constraint. Therefore, one may view the SEQ.1/SEQ.n fetch unit as identical to the trace processor frontend in Figure 3-2, except the trace cache block is replaced with a conventional instruction cache. That is, the next trace predictor drives a conventional instruction cache, and the trace buffers are used to construct "traces" from the L2 cache/main memory if not present in the cache.

Finally, to establish an upper bound on the performance of noncontiguous instruction fetching, I introduce a fourth model, TC-perfect, which is identical to TC but the trace cache always hits.

5.1.2 Isolating trace predictor/trace cache performance

An interesting side-effect of trace selection is that it significantly affects trace prediction accuracy. In general, smaller traces (resulting from more constrained trace selection) result in lower accuracy. There are at least two reasons for this. First, longer traces naturally capture longer path history. This can be compensated for by using more trace identifiers in the path history if the traces are small; that is, a good DOLC function for one trace length is not necessarily good for another. For the TC model, DOLC = {7,3,6,8} (a depth of 7 traces) consistently performs well over all benchmarks [39]. For SEQ.1 and SEQ.n, a brief search of the design space shows DOLC = {17,3,4,12} (a depth of 17 traces) performs well.

Tuning the DOLC parameters is apparently not enough, however -- trace selection affects accuracy in other ways. The graph in Figure 5-1 shows trace predictor performance using an unbounded table, i.e. using full, unhashed path history to make predictions. The graph shows trace mispredictions per 1000 instructions for SEQ.1, SEQ.n, and TC trace selection, as the history depth is varied. For the *go* benchmark, trace mispredictions for the SEQ.n model do not dip below 8.8 per 1000 instructions, whereas the TC model reaches as few as 8.0 trace mispredictions per 1000 instructions. Unconstrained trace selection results in the creation of many unique traces. While this trace explosion generally increases conflicts in the trace cache, I hypothesize it also creates many more unique contexts for making predictions. A large prediction table can exploit this additional context.

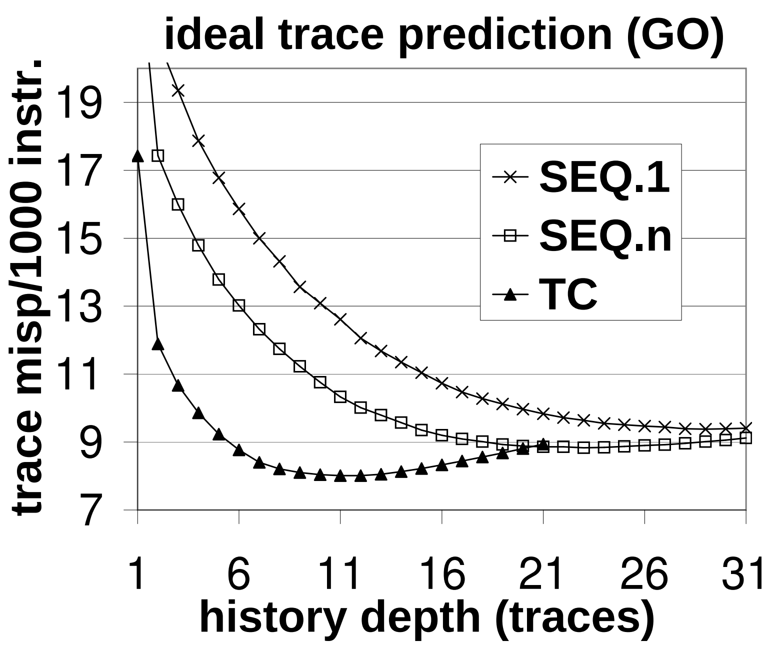


Figure 5-1: Impact of trace selection on unbounded trace predictor performance.

I conclude that it is difficult to separate the performance advantage of the trace cache from that of the trace predictor, because both show positive improvement with longer traces. Nonetheless, when I compare TC to SEQ.n or SEQ.1, I would like to know how much benefit is derived from the trace cache itself.

To this end, a methodology is developed to statistically “adjust” the overall branch prediction accuracy of a given fetch model to match that of another model. The trace predictor itself is not adjusted -- it produces predictions in the normal fashion. However, after making a prediction, the predicted trace is compared with the *actual* trace, determined in advance by a functional simulator running in parallel with the timing simulator. If the prediction is incorrect, the actual trace is substituted for the mispredicted trace *with some probability*. In other words, some fraction of mispredicted traces are corrected. The probability for injecting corrections was chosen on a per-benchmark basis to achieve the desired branch misprediction rate.

This methodology introduces two additional fetch models, SEQ.1-adj and SEQ.n-adj, corresponding to the “adjusted” SEQ.1 and SEQ.n models. Clearly these models are unrealizable, but they are useful for performance comparisons because their adjusted branch misprediction rates match that of the TC model.

5.1.3 Results

5.1.3.1 Performance of fetch models

Figure 5-2 shows the performance of the six fetch models in terms of retired instructions per cycle (IPC). The TC model in this section uses a 64KB (instruction storage only), 4-way set-associative trace cache. The trace cache is indexed using only the PC (i.e. no explicit path associativity, except that afforded by the 4 ways).

I can draw several conclusions from the graph in Figure 5-2. First, comparing the SEQ.n models to the SEQ.1 models, it is apparent that predicting and fetching multiple *sequential* basic blocks provides a significant performance advantage over conventional single-block fetching. The graph in Figure 5-3 shows that the performance advantage of the SEQ.n model over the SEQ.1 model ranges from about 5% to 25%, with the majority of benchmarks showing greater than 15% improvement. Similar results hold whether or not branch prediction accuracy is adjusted for the SEQ.n and SEQ.1 models.

This first observation is important because the SEQ.n model only requires a more sophisticated, high-level control flow predictor, and retains a more-or-less conventional instruction cache microarchitecture.

Second, the ability to fetch multiple, possibly *noncontiguous* basic blocks improves performance significantly over sequential-only fetching. The graph in Figure 5-4 shows that the performance advantage of the TC model over the SEQ.n model ranges from 15% to 35%.

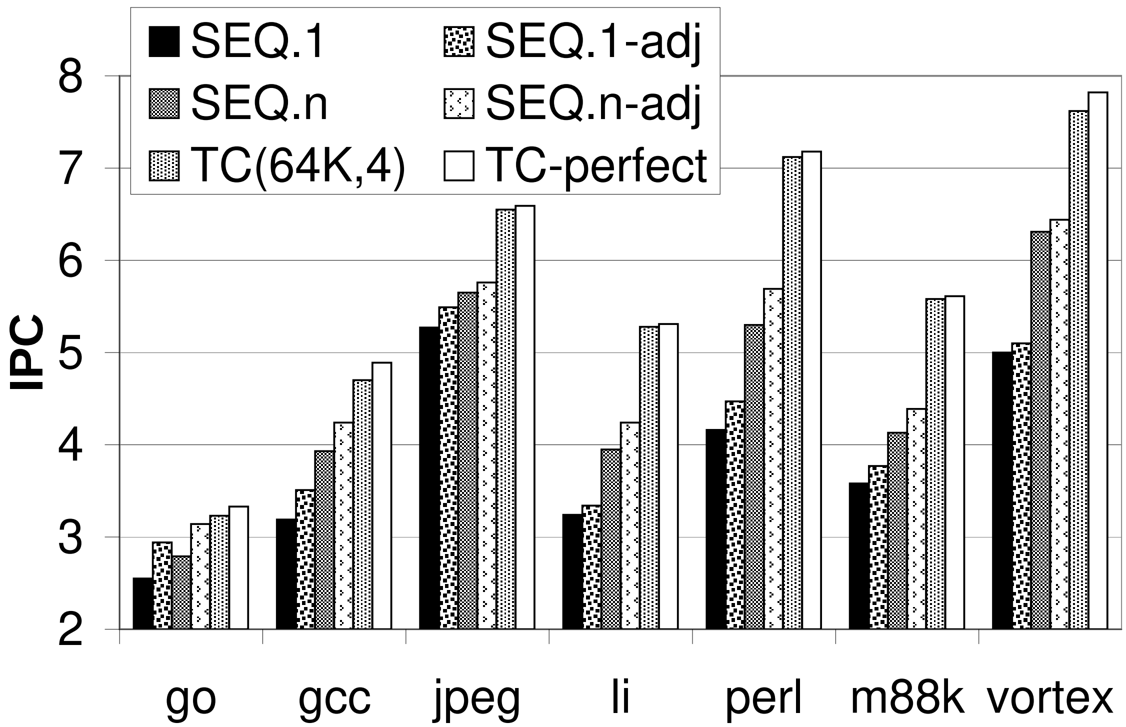


Figure 5-2: Performance of the fetch models.

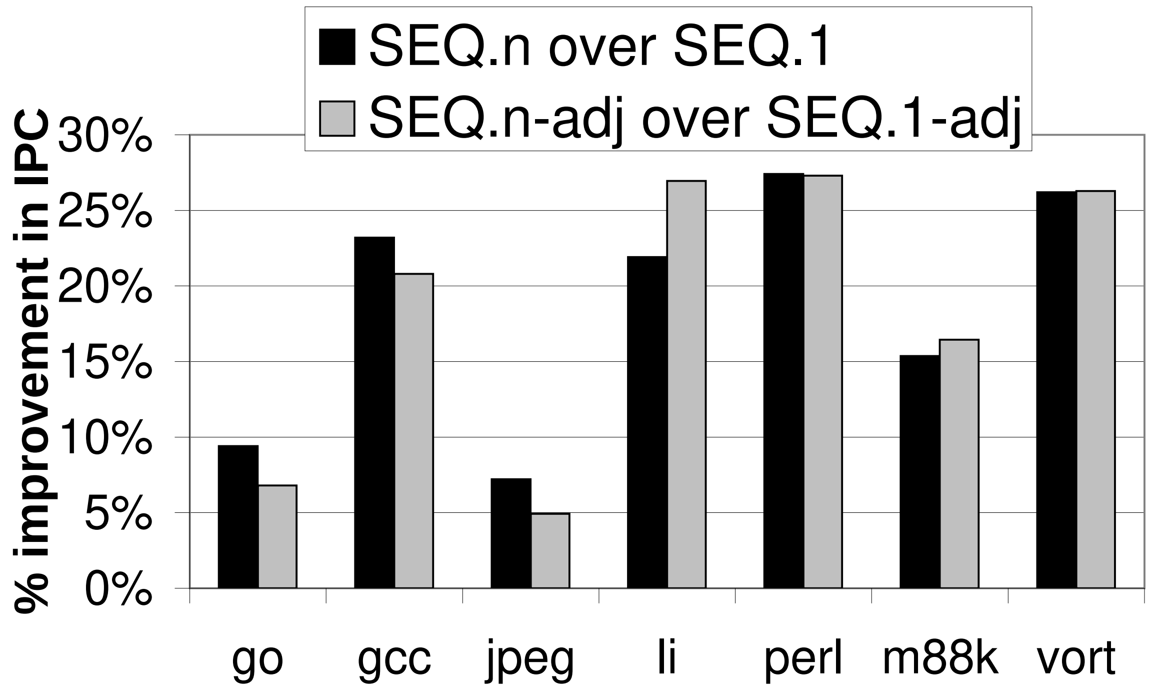


Figure 5-3: Speedup of SEQ.n over SEQ.1.

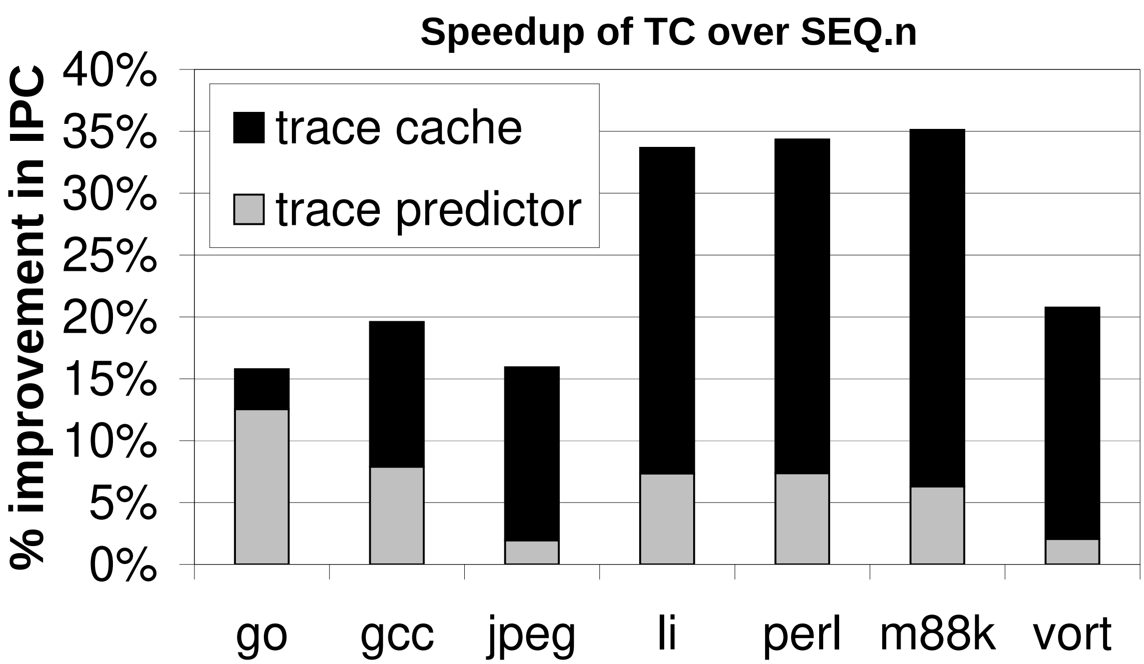


Figure 5-4: Speedup of TC over SEQ.n.

Figure 5-4 also isolates the contributions of next trace prediction and the trace cache to performance. The lower part of each bar is the speedup of model SEQ.n-adj over SEQ.n. And since the overall branch misprediction rate of SEQ.n-adj is adjusted to match that of the TC model, this part of the bar approximately isolates the impact of next trace prediction on performance. The top part of the bar therefore isolates the impact of the trace cache on performance.

For *go*, which suffers significantly more branch mispredictions than other benchmarks, most of the benefit of the TC model comes from next trace prediction. In this case, the longer traces of the TC model are clearly more valuable for improving the context used by the next trace predictor than for providing raw instruction bandwidth. For *gcc*, however, both next trace prediction and the trace cache contribute equally to performance. The other five benchmarks benefit mostly from higher fetch bandwidth.

Finally, Figure 5-2 shows the moderately large trace cache of the TC model very nearly reaches the performance upper bound established by TC-perfect (within 4%).

Table 5-1 shows trace- and branch-related measures. Average trace lengths for TC range from 12.4 (*li*) to 15.8 (*jpeg*) instructions (1.6 to over 2 times longer than SEQ.n traces).

The table also shows predictor performance: primary and alternate trace mispredictions per 1000 instructions, and overall branch misprediction rates (the latter is computed by checking each branch at retirement to see if it caused a misprediction, whether originating from the trace predictor or second-level branch predictor). In all cases prediction improves with longer traces. TC has from 20% to 45% fewer trace mispredictions than

SEQ.1, resulting in 15% (*jpeg*) to 41% (*m88ksim*) fewer total branch mispredictions. Note that the *adjusted* branch misprediction rates for the SEQ models are nearly equal to those of TC.

Table 5-1: Trace statistics.

model	measure	gcc	go	jpeg	li	m88k	perl	vort
SEQ.1	trace length	4.9	6.2	8.3	4.2	4.8	5.1	5.8
	trace misp./1000	8.8	14.5	5.2	6.9	3.5	3.4	1.5
	alt. trace misp./1000	2.1	4.5	0.1	0.6	0.4	0.1	0.2
	branch misp. rate	5.0%	11.0%	7.7%	3.7%	2.2%	2.2%	1.1%
	adjusted misp. rate	3.6%	8.2%	6.6%	3.2%	1.3%	1.4%	0.8%
SEQ.n	trace length	7.2	8.0	9.6	6.3	6.0	7.1	8.2
	trace misp./1000	7.3	12.7	4.6	6.9	3.3	3.1	1.2
	alt. trace misp./1000	2.7	5.4	0.5	0.9	0.6	0.3	0.3
	branch misp. rate	4.4%	10.1%	7.0%	3.7%	2.1%	2.0%	0.9%
	adjusted misp. rate	3.6%	8.1%	6.7%	3.1%	1.3%	1.4%	0.8%
TC	trace length	13.9	14.8	15.8	12.4	13.1	13.0	14.4
	trace misp./1000	5.4	9.6	4.2	5.5	2.0	2.1	1.0
	alt. trace misp./1000	2.7	5.3	0.9	1.3	0.5	0.3	0.3
	branch misp. rate	3.6%	8.2%	6.7%	3.1%	1.3%	1.5%	0.8%
	control instr. per trace	2.8	2.3	1.3	2.9	2.5	2.5	2.3
	$RF_{overall}$	7.1	14.4	5.3	3.1	3.7	4.1	2.9
	RF_{dyn}	3.0	3.3	3.7	3.2	3.1	2.9	2.1

Shorter traces, however, generally result in better alternate trace prediction accuracy. Shorter traces result in 1) fewer total traces and thus less aliasing, and 2) fewer possible alternative traces from a given starting PC. For all benchmarks except *gcc* and *go*, the alternate trace prediction is almost always correct given the primary trace prediction is incorrect -- both predictions taken together result in fewer than 1 trace misprediction per 1000 instructions.

Trace caches introduce redundancy -- the same instruction can appear multiple times in one or more traces. Table 5-1 shows two redundancy measures. The *overall redundancy factor*, $RF_{overall}$, is computed by maintaining a table of all unique traces ever retired. Redundancy is the ratio of total number of instructions to total number of *unique* instructions for traces collected in the table. $RF_{overall}$ is independent of trace cache configuration and does not capture dynamic behavior. The *dynamic redundancy factor*, RF_{dyn} , is computed similarly, but using only traces *in the trace cache in a given cycle*; the final value is an average over all cycles. RF_{dyn} was measured using a 64KB, 4-way trace cache.

$RF_{overall}$ varies from 2.9 (*vortex*) to 14 (*go*). RF_{dyn} is less than $RF_{overall}$ and only ranges between 2 and 4, because the fixed size trace cache limits redundancy, and perhaps temporally there is less redundancy.

5.1.3.2 Trace cache size and associativity

In this section, I measure performance of the TC model as a function of trace cache size and associativity. Figure 5-5 shows overall performance (IPC) for 12 trace cache configurations: direct mapped, 2-way, and 4-way associativity for each of four sizes, 16KB, 32KB, 64KB, and 128KB.

Associativity has a noticeable impact on performance for all of the benchmarks except *go*. *Go* has a particularly large working set of unique traces [81], and total capacity is more important than individual trace conflicts. The curves of *jpeg* and *li* are fairly flat -- size is of little importance, yet increasing associativity improves performance. These two benchmarks suffer few general conflict misses (otherwise size should improve performance), yet conflicts among traces with the same start PC are significant. Associativity allows simultaneously caching these path-associative traces.

The performance improvement of the largest configuration (128KB, 4-way) with respect to the smallest one (16KB, direct mapped) ranges from 4% (*go*) to 10% (*gcc*).

Figure 5-6 shows trace cache performance in *misses per 1000 instructions*. Trace cache size is varied along the x-axis, and there are six curves: direct mapped (DM), 2-way (2W), and 4-way (4W) associative caches, both with and without indexing for path associativity (PA). The following index function was chosen (somewhat arbitrarily) for achieving path associativity: the low-order bits of the PC form the set index, and then the high-order bits of this index are XORed with the first two branch outcomes of the trace identifier.

Gcc and *go* are the only benchmarks that do not fit entirely within the largest trace cache. As I observed earlier, *go* has many heavily-referenced traces, resulting in no fewer than 20 misses/1000 instructions.

Path associativity reduces misses substantially, particularly for direct mapped caches. Except for *vortex*, path associativity closes the gap between direct mapped and 2-way associative caches by more than half, and often entirely.

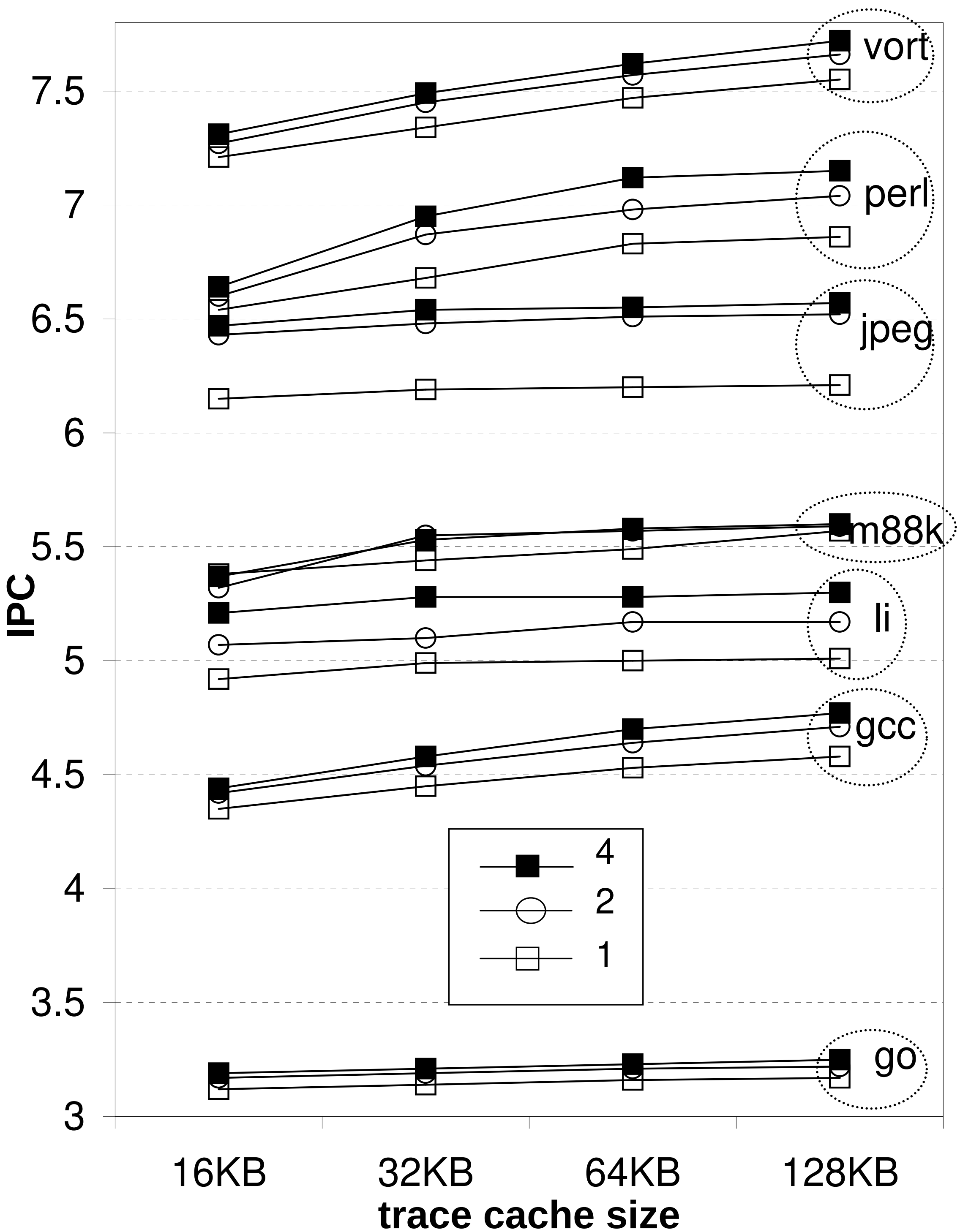


Figure 5-5: Performance vs. size/associativity.

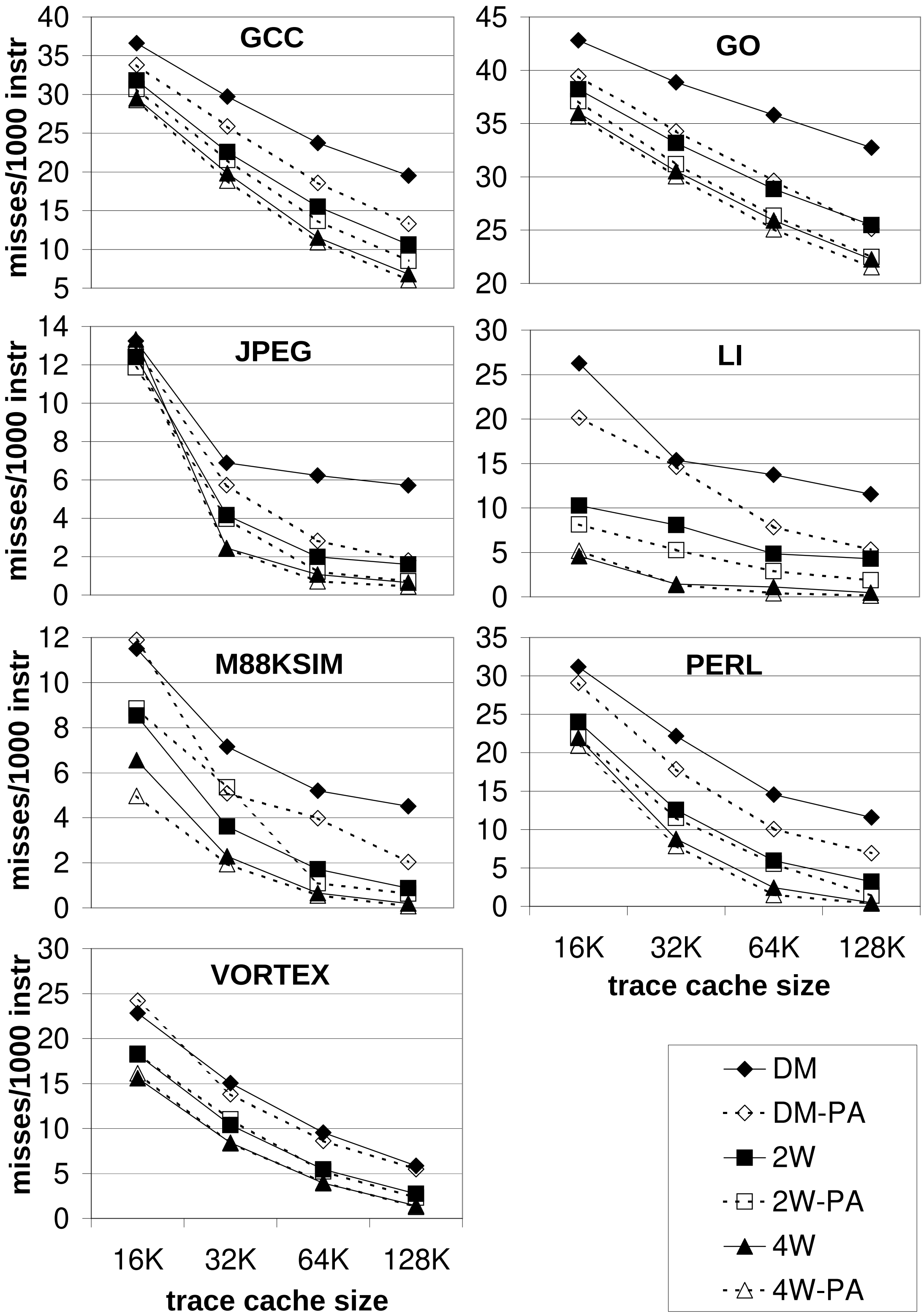


Figure 5-6: Trace cache misses.

5.2 Comparative study of superscalar and trace processors

Trace processors present a tradeoff between complexity and instructions executed per cycle (IPC). Ignoring complexity and cycle time, the trace processor is outperformed by its superscalar counterpart in terms of IPC, for two reasons. Firstly, *load balance* becomes an issue in a distributed implementation, and secondly, *partial bypassing* increases the latency (in clock cycles) to resolve global data dependences. On the other hand, distributed resources and partial bypasses potentially enable a faster clock.

To be a viable microarchitecture alternative, the trace processor's cycle time advantage must outweigh its IPC limitations. In this section, a comparative study of superscalar and trace processors is developed to better understand the IPC/cycle time tradeoff. The study is organized in three parts.

1. Section 5.2.2 quantifies the performance impact of distributing the instruction window. I first explain three load balance factors that degrade trace processor performance. Then, their combined impact is measured by comparing superscalar and trace processors of equal window size and issue bandwidth. To isolate only the impact of distributed resources, both superscalar and trace processors use full bypassing.

2. Section 5.2.3 compares superscalar processors and realistic trace processors employing partial bypassing. The trace processor incurs one full cycle to bypass global values.

   Penalizing global communication in the trace processor provides a simple abstraction for reasoning about the relative complexity of the two processor models. Since I account for "global" complexity via a one cycle penalty, overall trace processor complexity can be approximately characterized by PE, or “local”, complexity -- something one can more directly reason about.

I compare superscalar and trace processors as follows. First, I identify superscalar and trace processor configurations that give similar IPC performance. Then, I qualitatively reason about cycle time based on the relative complexity of a single PE versus the full superscalar processor.

3. Quantifying complexity is not as straightforward as measuring IPC. Fortunately, Palacharla, Jouppi, and Smith [69] derived techniques for analyzing relative complexity in superscalar processors. In Section 5.2.4, their analytical results are applied to several design points in a limited demonstration of the trace processor’s *overall performance advantage*.

5.2.1 Experimental setup

The superscalar and trace processor models are virtually identical, except 1) the superscalar processor has a single, centralized instruction window and 2) the trace processor requires one full cycle to bypass global values, unless stated otherwise.

The superscalar processor uses the same hierarchical frontend as the trace processor, and it also benefits from memory dependence speculation. And like the superscalar processor, the trace processor is provided as many (global) result buses as its aggregate issue bandwidth, i.e., result bus bandwidth is not a performance limiter.

Full analysis of global result bus bandwidth is deferred to Section 5.4, where I show the number of required buses is reasonable (e.g. 4 to 6 for large trace processors). Cycle time estimation in Section 5.2.4 requires specifying the actual number of global result buses, in which case I specify the number of buses that performs as well as unconstrained buses.

I simulated a wide range of superscalar and trace processor configurations. Configurations are labeled as follows.

- superscalar processor:

SS-<*window size*>-<*issue width*>

- trace processor:

TP-<*total window size*>-<*total issue width*> (<*trace length*>/<*# PEs*>/<*issue per PE*>)

-OR-

<*trace length*>/<*# PEs*>

All trace processor configurations in Section 5.2 have a maximum trace length of 16 instructions, which is also the size of each PE window. The number of PEs is 2, 4, 8, or 16, and the issue bandwidth is 1-, 2-, or 4-way issue per PE. As indicated above, trace processors are labeled both with aggregate parameters and per-PE parameters; <*total window size*> is equal to <*# PEs*> times <*trace length*>, and <*total issue width*> is equal to <*# PEs*> times <*issue per PE*>. For graphs and tables that vary PE issue width along an axis or column, configurations are labeled with the shorthand notation indicated above (trace length and number of PEs).

Superscalar configurations are listed in Table 5-2. For small superscalar processors, the trace length is less than 16 instructions. Shorter traces were chosen to improve trace cache hit rate where higher bandwidth is not needed.

Table 5-2: Superscalar processor configurations.

	SS-16	SS-32	SS-64	SS-128	SS-256
trace length	4	8	16	16	16
issue width	1, 2, 4	1, 2, 4, 8	2, 4, 8, 16	4, 8, 16	4, 8, 16

5.2.2 Performance impact of distributing the instruction window

Ignoring complexity and cycle time, a centralized dynamic scheduling window will outperform an equivalent distributed one due to the following three reasons.

1. *Distributed issue bandwidth.*

   The trace processor divides the instruction window into equal partitions (PEs) and dedicates an equal slice of the total issue bandwidth to each partition. One PE may have more instructions ready to issue than its allotted bandwidth while another PE has idle issue slots. The trace processor is inflexible and cannot remedy the load imbalance.

2. *Window fragmentation.*

   The trace selection algorithm may produce traces shorter than the maximum trace length, wasting instruction issue buffers and execution bandwidth in the PE. Although the superscalar model also uses traces, it has a single, large instruction window that does not suffer fragmentation due to short traces.

3. *Discrete window management*.

   In the trace processor, instruction buffers are freed in discrete chunks. That is, a PE is freed and available for allocation only when *all* instructions in the current trace have completed. Discrete window management is inefficient because instruction buffers that would otherwise be available are tied up by other instructions in the same trace. The effect is amplified in trace processors with very long traces (e.g. 32) and a small number of PEs (e.g. 2).

Distributed issue bandwidth, window fragmentation, and discrete window management are all grouped under the more general term “load balance”.

The graphs in figures 5-9 through 5-16 (one for each benchmark) plot the performance of all processor configurations. Dashed and solid curves correspond to superscalar and trace processors, respectively. Each curve corresponds to a single (aggregate) window size. Superscalar issue bandwidth, or issue bandwidth per PE for trace processors, is varied along the x-axis. Thus, the graph shows how performance (IPC) scales with issue bandwidth and window size.

As mentioned earlier in the outline of Section 5.2, for the purposes of this study, the trace processor is not penalized an extra cycle for bypassing global values. I wish to isolate only the performance impact of load balance.

Load balance is quantified by comparing “equivalent” superscalar and trace processors. Two configurations are equivalent if they have the same aggregate window size and issue bandwidth. For example, consider what happens to performance when the instruction window and issue bandwidth of SS-64-4 are divided equally into four parts, resulting in the equivalent TP-64-4(16/4/1) configuration. The IPC of *gcc* decreases from about 2.75 to just under 2.0, a decrease of 30%.

The performance of nine trace processors relative to their superscalar counterparts is summarized in Figure 5-7 for all of the benchmarks. Clearly, a distributed instruction window performs worse than a centralized instruction window of equal size and bandwidth. Over all benchmarks, distributing the instruction window results in a performance loss of 20% to 30% for most configurations.

Load balance is more pronounced for 8 and 16 single-issue PEs. TP-128-8(16/8/1) and TP-256-16(16/16/1) perform 40% to 50% worse than SS-128-8 and SS-256-16, respectively. These configurations have large instruction windows and expose a significant amount of instruction-level parallelism. Unfortunately, single-issue PEs are too inflexible and largely waste this potential. On the other hand, simply increasing PE issue bandwidth from 1-way to 2-way issue significantly improves load balance.

Two trends are evident from Figure 5-7. For large instruction windows, load balance effects are pronounced at narrow PE issue widths but this is remedied by increasing width. For small instruction windows, namely 2 PEs, load balance is significant and relatively insensitive to PE issue width. To help interpret this data, I isolate the combined performance impact of the two factors that effectively *reduce the window size of the trace processor*: discrete window management and window fragmentation. The distributed issue bandwidth factor is eliminated by giving the trace processor a centralized issue bandwidth constraint (like superscalar). From the resulting graph in Figure 5-8, the performance of 2 PEs with respect to equivalent superscalar processors steadily decreases with increasing PE issue rate. Clearly, eliminating the distributed issue bandwidth factor only exposes the other two factors. And as might be expected, this is mostly true for small windows (2 PEs) -- the effective reduction in window size due to discrete window management and fragmentation is relatively significant for small windows. For example, discrete window management potentially halves the window size with 2 PEs by rendering the least speculative PE underutilized.

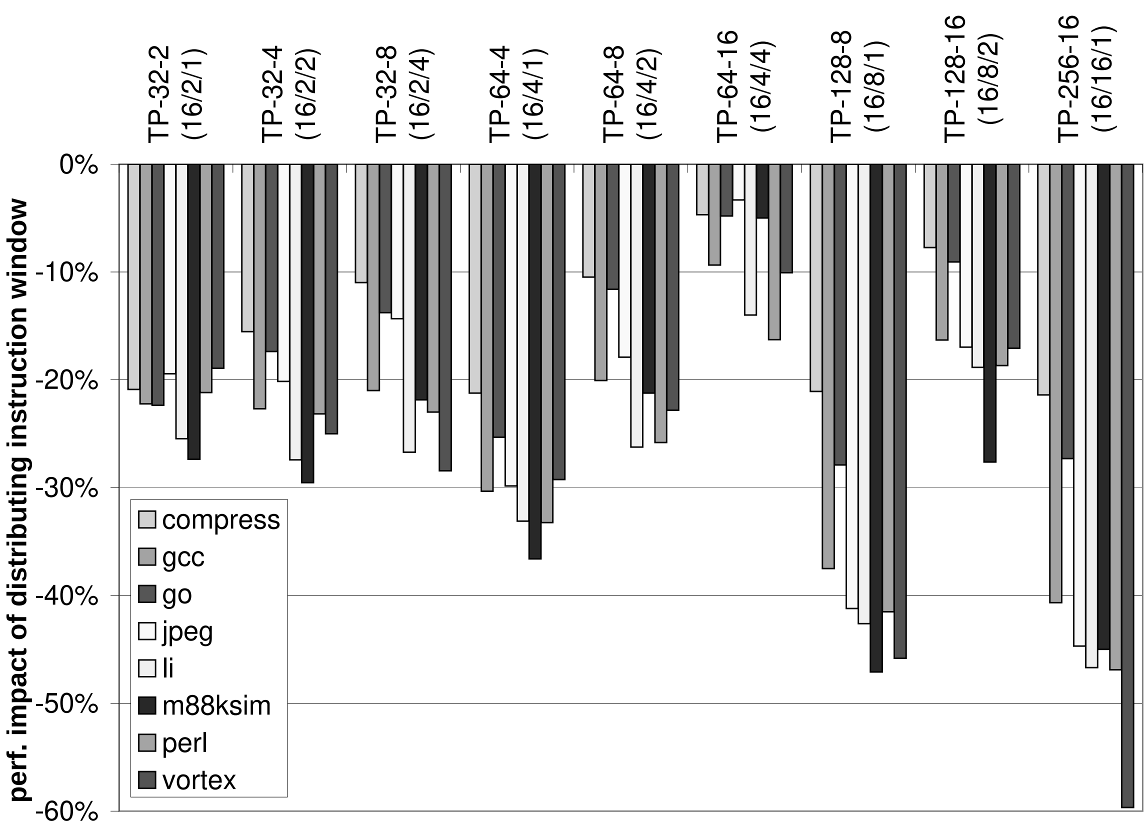


Figure 5-7: Performance impact of distributing the instruction window.

Comparing processors on the basis of equal window size and issue width is somewhat arbitrary, contrived to point out the problem of load balance. There are two variables in the overall performance equation: IPC and cycle time. Therefore, it is more useful to compare two processors that have either equivalent IPC or equivalent complexity. In the next subsection, processors having similar performance (IPC) are identified, and then I make qualitative observations about the relative complexity of these processors to arrive at overall performance.

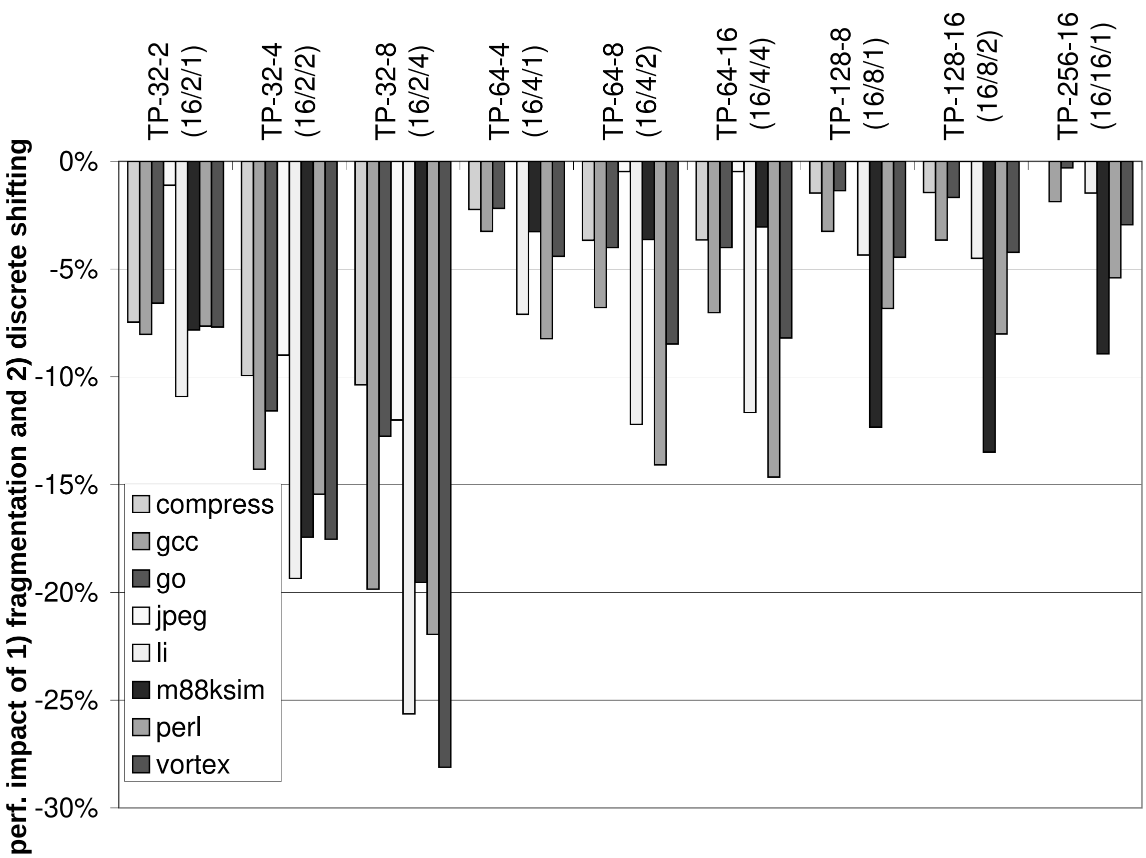


Figure 5-8: Impact of window fragmentation and discrete window management.

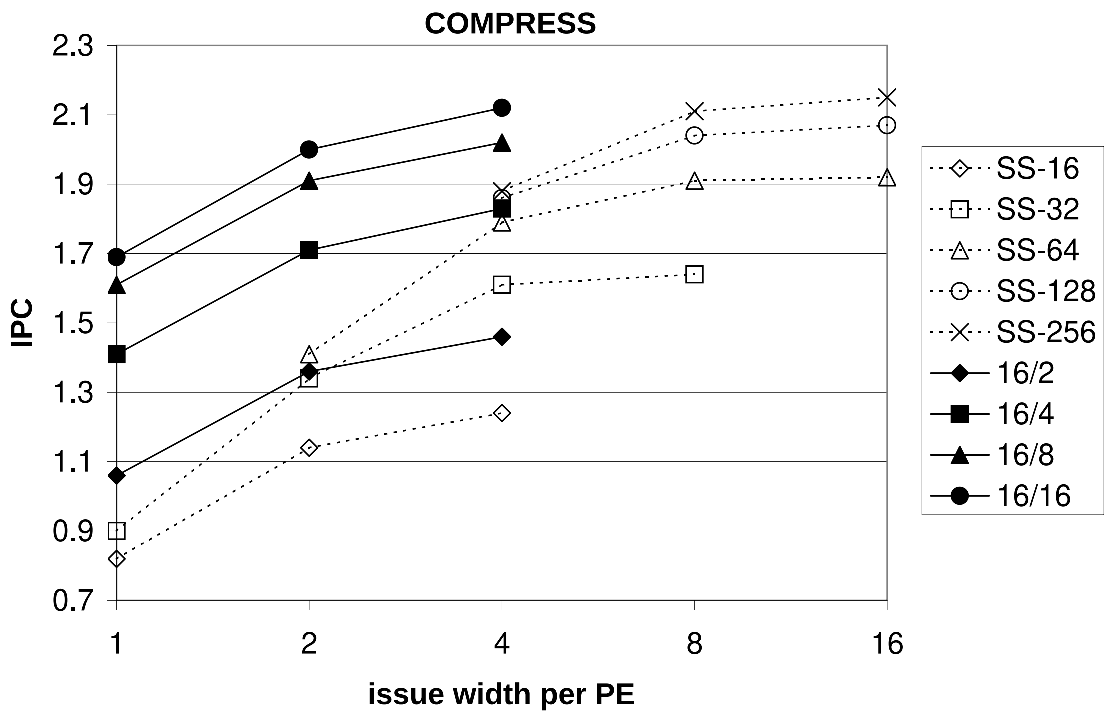


Figure 5-9: Superscalar vs. trace processors with ideal bypasses (*compress*).

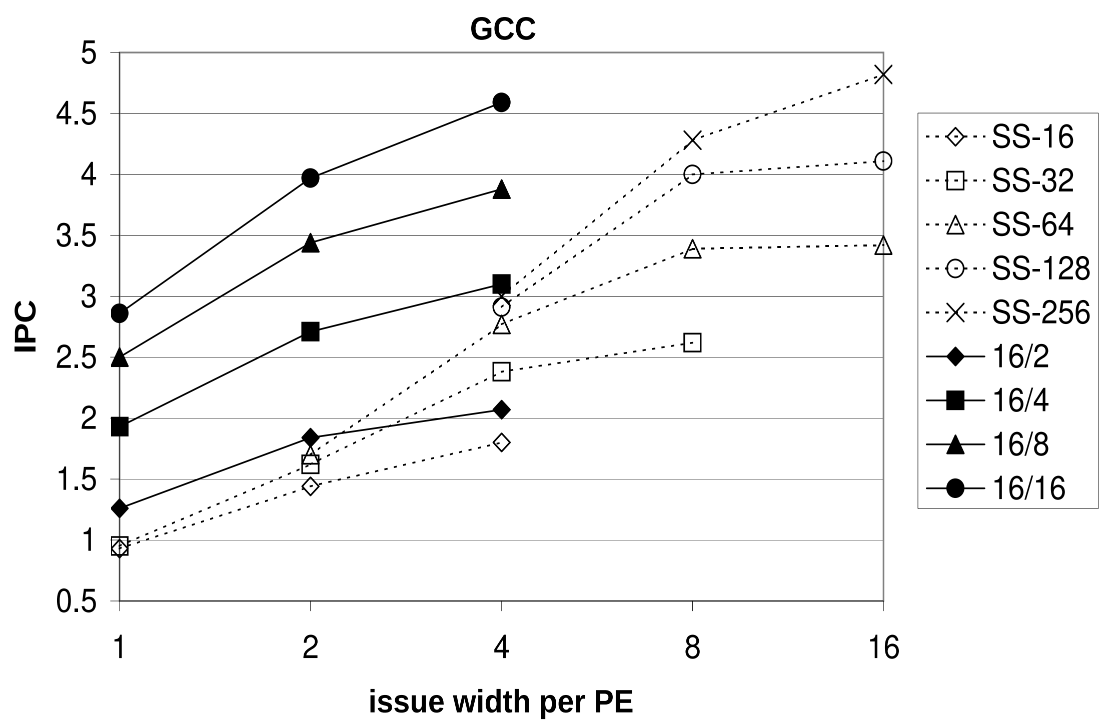


Figure 5-10: Superscalar vs. trace processors with ideal bypasses (*gcc*).

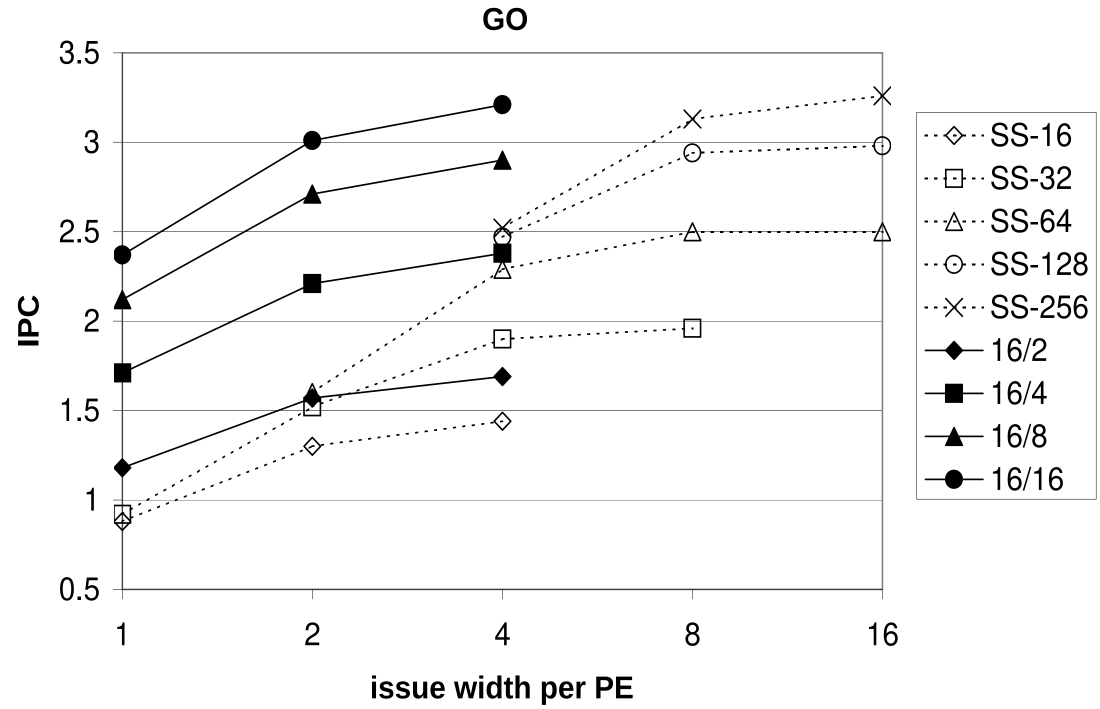


Figure 5-11: Superscalar vs. trace processors with ideal bypasses (*go*).

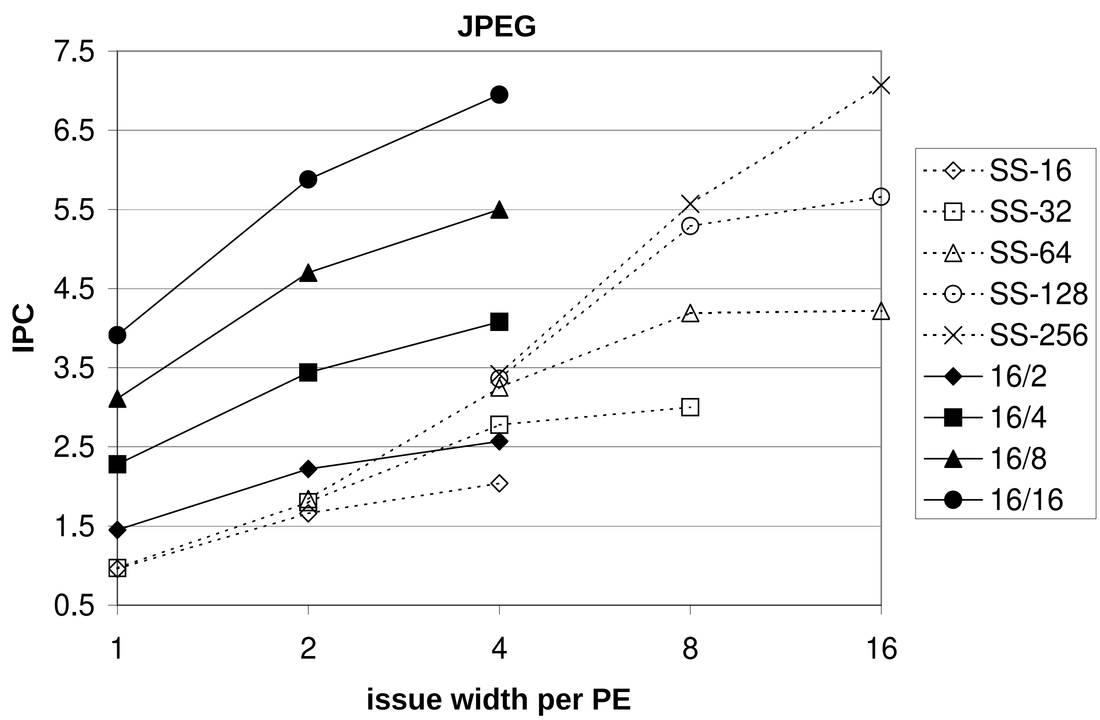


Figure 5-12: Superscalar vs. trace processors with ideal bypasses (*jpeg*).

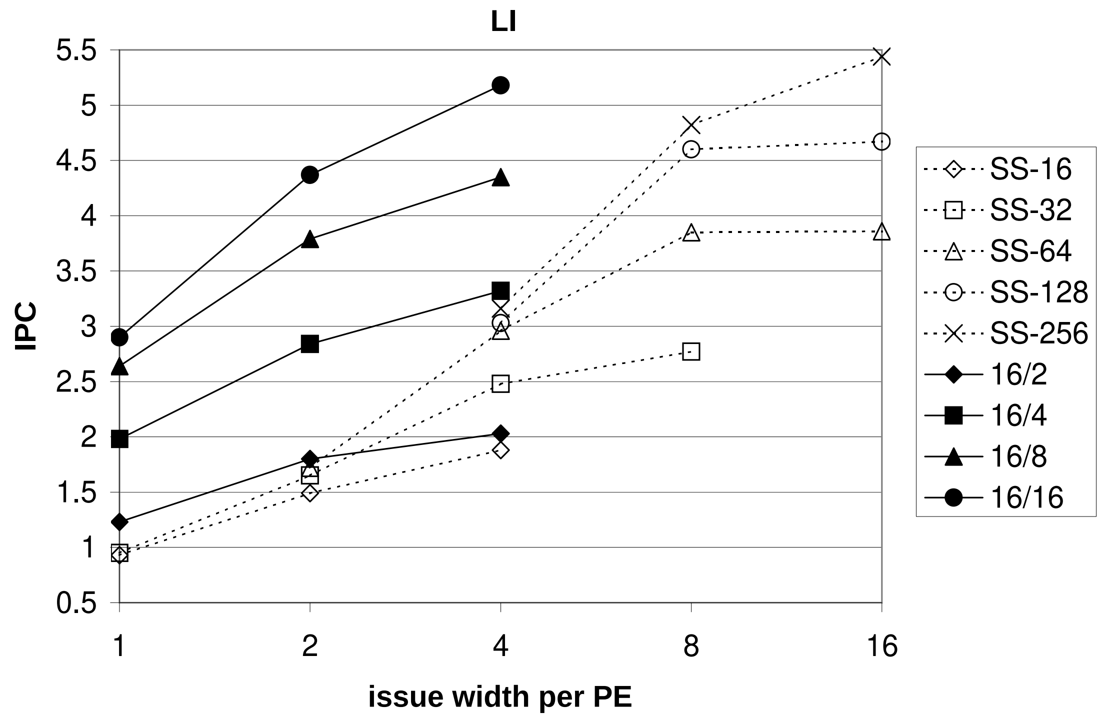


Figure 5-13: Superscalar vs. trace processors with ideal bypasses (*li*).

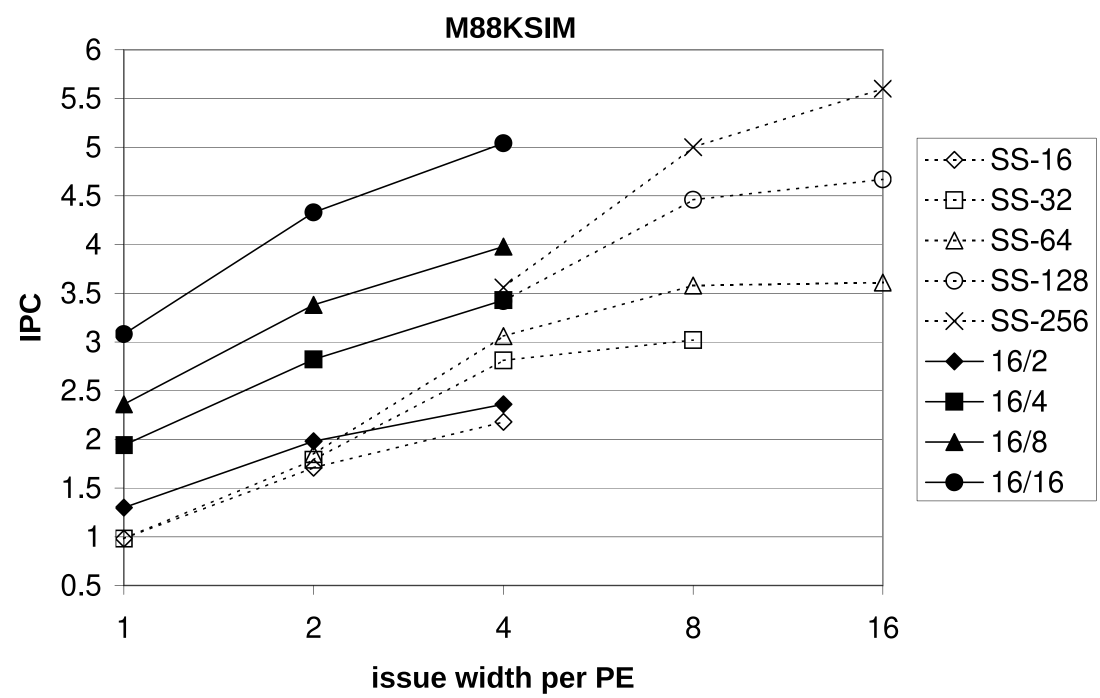


Figure 5-14: Superscalar vs. trace processors with ideal bypasses (*m88ksim*).

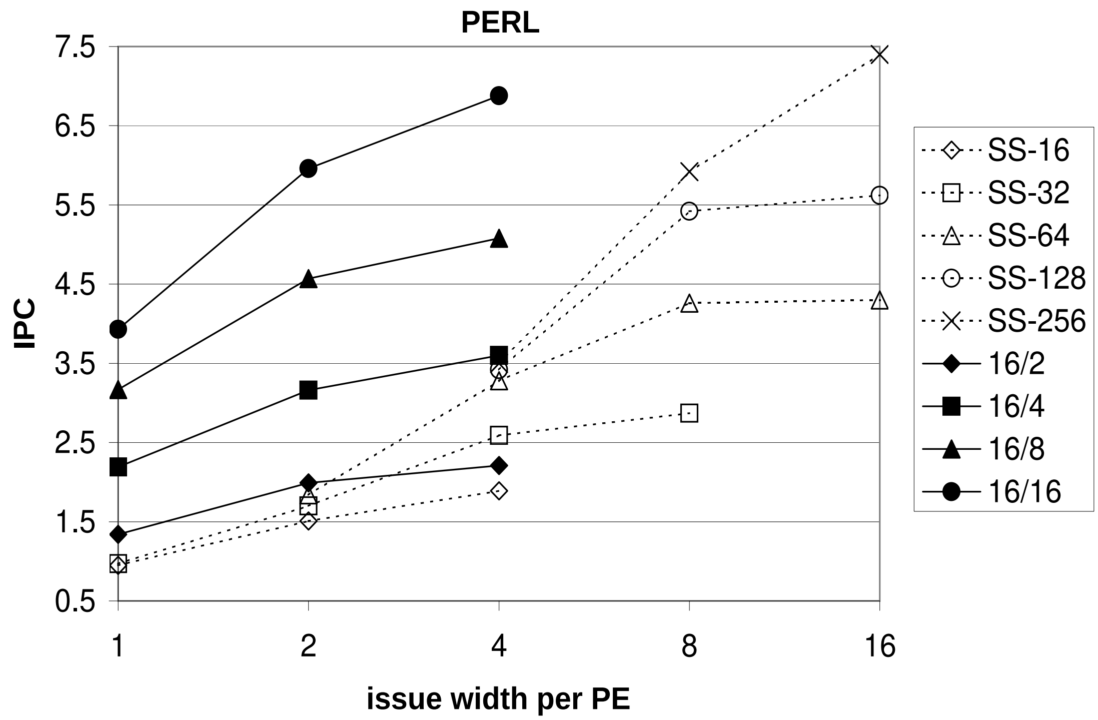


Figure 5-15: Superscalar vs. trace processors with ideal bypasses (*perl*).

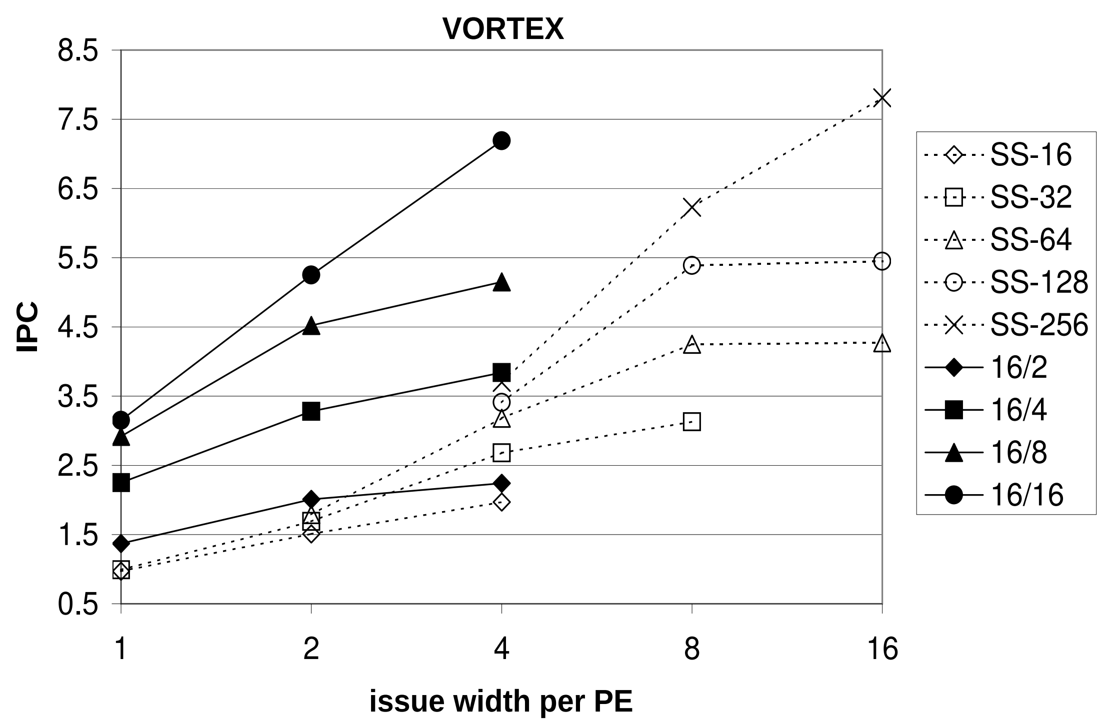


Figure 5-16: Superscalar vs. trace processors with ideal bypasses (*vortex*).

5.2.3 Analysis of similar-performance superscalar and trace processors

In Figures 5-17 through 5-24, two graphs are presented for each benchmark. The first graph plots the performance of all superscalar and trace processor configurations, in the same manner described previously in Section 5.2.2. The only difference between these graphs and those in the previous subsection is that *the trace processor requires one full cycle to bypass global values*.

In the second graph of each figure, processor configurations that have similar performance are grouped together into “similar-IPC groups”. The purpose of this graph is to quickly identify superscalar and trace processor configurations that perform equally well.

For example, a 4 PE trace processor with single-issue elements (`TP-64-4(16/4/1)`) and a 4-way superscalar processor with a 16 instruction window (`SS-16-4`) have similar performance for *gcc*, *jpeg*, *li*, *perl*, and *vortex*. Both processors have an aggregate issue bandwidth of 4. Despite partitioning its issue bandwidth, `TP-64-4` manages to utilize the bandwidth equally well. This requires an overall larger window (4 times larger). The circuit complexity of the trace processor is approximately that of a single PE, however, since global complexity is accounted for by partial bypassing. A PE has complexity on the order of `SS-16-1` and can potentially be clocked faster than `SS-16-4`.

The results in Figures 5-18 through 5-24 are summarized in Table 5-3. The table pairs each trace processor with a similar-IPC superscalar processor. (In some cases, there is no superscalar processor with reasonably similar performance: this appears as a blank table entry.) The pairing is performed for each benchmark. Fortunately, there is usually a dominant pairing across all benchmarks. The majority pairing is indicated in the "Summary" column. The summary column allows comparisons to be made relatively independent of benchmark.

If there are multiple superscalar configurations that can be paired with a trace processor, in general, I choose the less complex configuration based on my understanding of analytical models to be given in Section 5.2.4. In some cases, there are (+) and (-) indicators in a table entry. A (+) denotes the trace processor performs noticeably better than the indicated superscalar processor, but the IPCs are similar enough to pair. A (-) denotes the reverse is true.

Below I discuss the trace processor/superscalar processor pairings from Table 5-3.

- `16/2` trace processors

  The smallest trace processor configurations, `TP-32-2(16/2/1)` and `TP-32-4(16/2/2)`, show encouraging results. They perform as well as superscalar processors with a 16 instruction window and 2-way or 4-way issue, i.e. `SS-16-2` and `SS-16-4`, respectively. The superscalar and trace processors have equal aggregate issue bandwidth, but individual PEs have the same size window and half the issue width of corresponding superscalar processors.

  Two small superscalar processors can be connected together to achieve the same IPC performance as doubling the issue width of one processor. Furthermore, both the perceived and real simplicity of replicating PEs provides a better approach to increasing hardware parallelism.

Notice the `TP−32−8(16/2/4)` configuration is not an effective design point. Two 4-way issue PEs together perform little better than one PE (`SS−16−4`), presumably because 2-way issue sufficiently exploits the parallelism within a 16 instruction trace.

- `16/4` trace processors

4 PE trace processors continue the promising trends of 2 PE configurations. Connecting 4 single-issue PEs to form `TP−64−4(16/4/1)` gives the same or better performance as quadrupling the issue width of one processing element to form `SS−16−4`. Overall circuit complexity remains relatively fixed as the single-issue PEs are replicated and connected into a trace processor, whereas quadrupling superscalar issue bandwidth incurs significant additional complexity in the select logic and result bypass circuits, as will be shown in the next subsection.

Interestingly, the `TP−32−4(16/2/2)` and `TP−64−4(16/4/1)` trace processors perform similarly due to their equal aggregate issue bandwidth. The choice of one configuration over another is a tradeoff between adding more PEs and increasing the parallelism of a single PE. Except for *m88ksim*, `16/4/1` performs slightly better than `16/2/2` due to the larger overall instruction window (64 versus 32). The 4 PE trace processor is also less sensitive to discrete window management (although it is potentially more sensitive to other load balance factors).

`TP−64−8(16/4/2)` performs similarly to `SS−32−8` and, to a lesser extent, `SS−64−4`. To achieve similar performance, the trace processor matches and then doubles either the window size (`SS-32-8`) or the issue bandwidth (`SS-64-4`), but not both -- an encouraging result considering the trace processor incurs performance penalties due to load balance and partial bypassing. And the superscalar processors are considerably larger and wider than a single 2-way issue processing element.

The `TP-64-16(16/4/4)` trace processor is required to match the performance of `SS-64-4` in many of the benchmarks, however. Nevertheless, a 16x4 PE is less complex than a 64x4, 128x4, or 256x4 superscalar processor (in several benchmarks, the `16/4/4` trace processor performs similarly to `SS-128-4` and `SS-256-4`).

- `16/8` trace processors

The performance of `TP-128-8(16/8/1)` is often subsumed by a smaller trace processor with 2-way issue elements, i.e. TP-64-8(16/4/2). It is consequently a less interesting design point than the other 8-PE trace processors.

The `TP-128-16(16/8/2)` trace processor with 2-way issue PEs appears to be the most effective design point of the 8-PE configurations. This trace processor performs about the same as `SS-64-8`. The trace processor's aggregate window size and issue width are both twice that of the superscalar processor, but the complexity of a single PE is similar to `SS-16-2` -- a quarter the "dimensions" of `SS-64-8`.

In the next subsection, `TP-128-16(16/8/2)` and `SS-64-8` form the basis of my case study on quantitative complexity analysis.

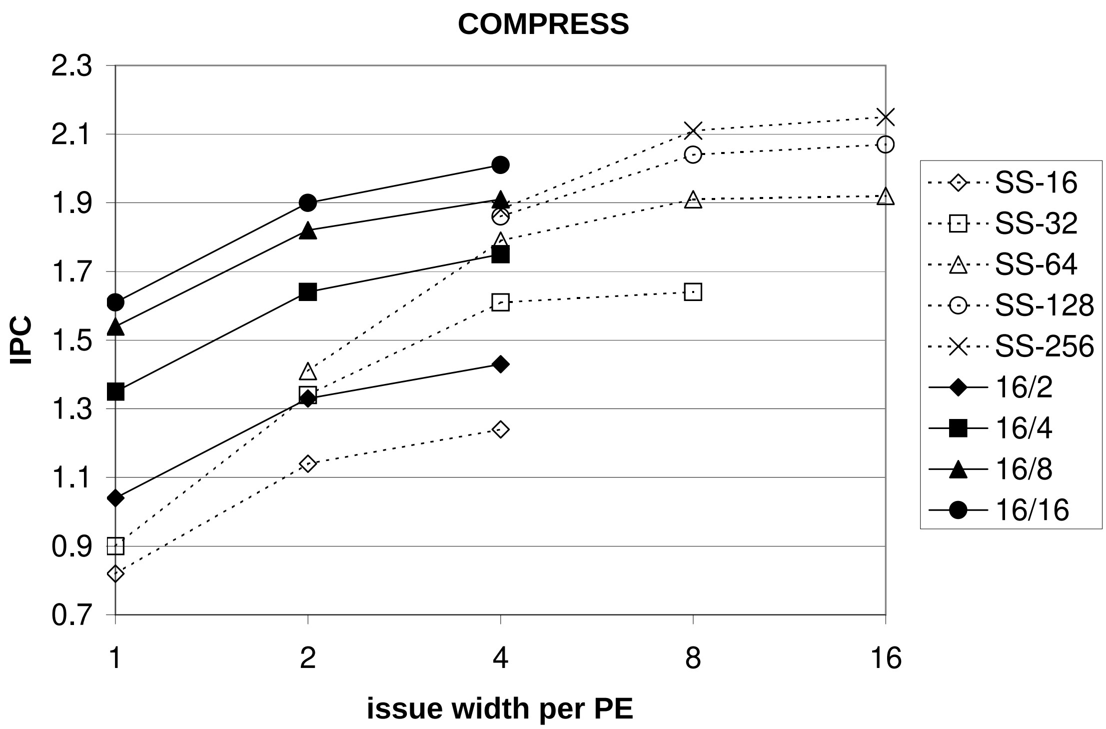
- `16/16` trace processors

  `TP-256-16(16/16/1)` may be more complexity-effective than `TP-64-16(16/4/4)` for equaling the performance of `SS-64-4`. `16/16/1` has single-issue PEs, whereas `16/4/4` -- a smaller processor overall -- requires 4-way issue per PE.

  The performance of the `16/16/2` and `16/16/4` processors is matched only by superscalar processors having 8-way or higher issue bandwidth. Although the transition from 1-way to 4-way issue incurs significant additional complexity, it is the leap to 8-way and higher issue rates that poses a serious problem. Consequently, the benefit of trace processors and other complexity-effective microarchitectures is undoubtedly greatest in the context of very wide issue processors.

Table 5-3: Pairing trace processors with similar-IPC superscalar processors.

Trace Proc.		compress	gcc	go	jpeg	li	m88ksim	perl	vortex	Summary
16/2	1		SS-16-2 (-)	SS-16-2 (-)	SS-16-2			SS-16-2	SS-16-2	SS-16-2
	2	SS-32-2	SS-16-4	SS-32-2	SS-16-4	SS-64-2	SS-64-2	SS-16-4	SS-16-4	SS-16-4
	4	SS-64-2		SS-64-2		SS-16-4	SS-16-4 (+)	SS-16-4 (+)	SS-16-4 (+)	SS-16-4
16/4	1	SS-32-2	SS-16-4	SS-64-2	SS-16-4	SS-16-4	SS-64-2	SS-16-4 (+)	SS-16-4 (+)	SS-16-4
	2	SS-32-4	SS-32-8	SS-32-8 (+)	SS-64-4	SS-32-8		SS-32-8	SS-64-4	SS-32-8
	4	SS-64-4	SS-128-4	SS-64-4		SS-64-4	SS-64-4	SS-64-4	SS-256-4	SS-64-4
16/8	1		SS-32-4	SS-32-8	SS-32-4	SS-32-4	SS-16-4	SS-32-8	SS-32-4	SS-32-4
	2	SS-64-4	SS-64-8 (-)	SS-128-4	SS-64-8		SS-64-4	SS-64-8	SS-64-8 (-)	SS-64-8
	4	SS-64-8	SS-64-8 (+)			SS-64-8 (+)	SS-64-8			SS-64-8
16/16	1	SS-32-4	SS-64-4	SS-64-4	SS-64-4 (+)	SS-32-4	SS-32-4	SS-64-4 (+)	SS-32-4	SS-64-4
	2	SS-64-8	SS-64-8 (+)	SS-128-8 (-)	SS-128-8	SS-64-8		SS-128-8	SS-64-8 (+)	SS-64-8
	4	SS-128-8	SS-256-8 (-)	SS-128-8		SS-128-8	SS-128-8	SS-256-8	SS-256-8	SS-128-8



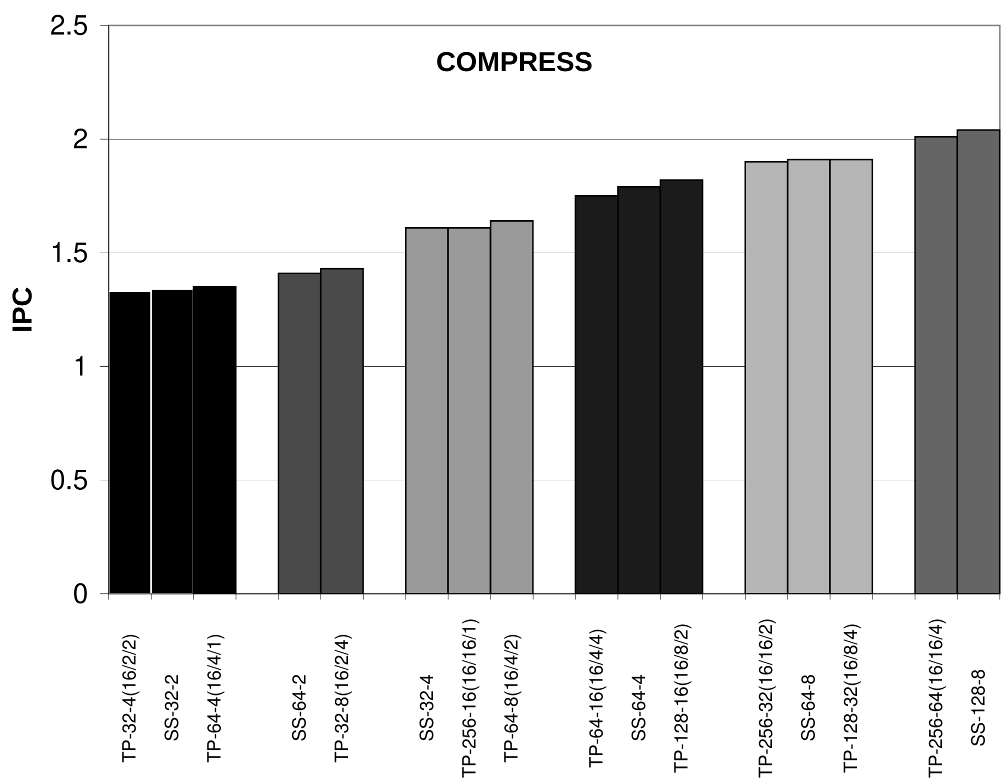
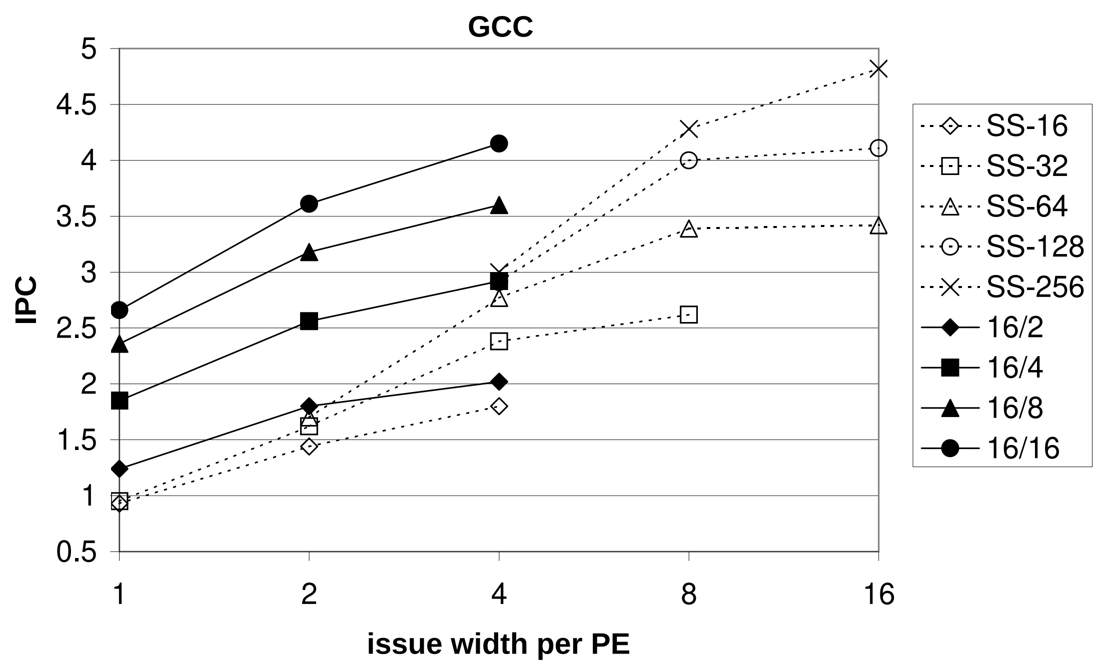


Figure 5-17: Superscalar vs. trace processors with partial bypasses (*compress*).



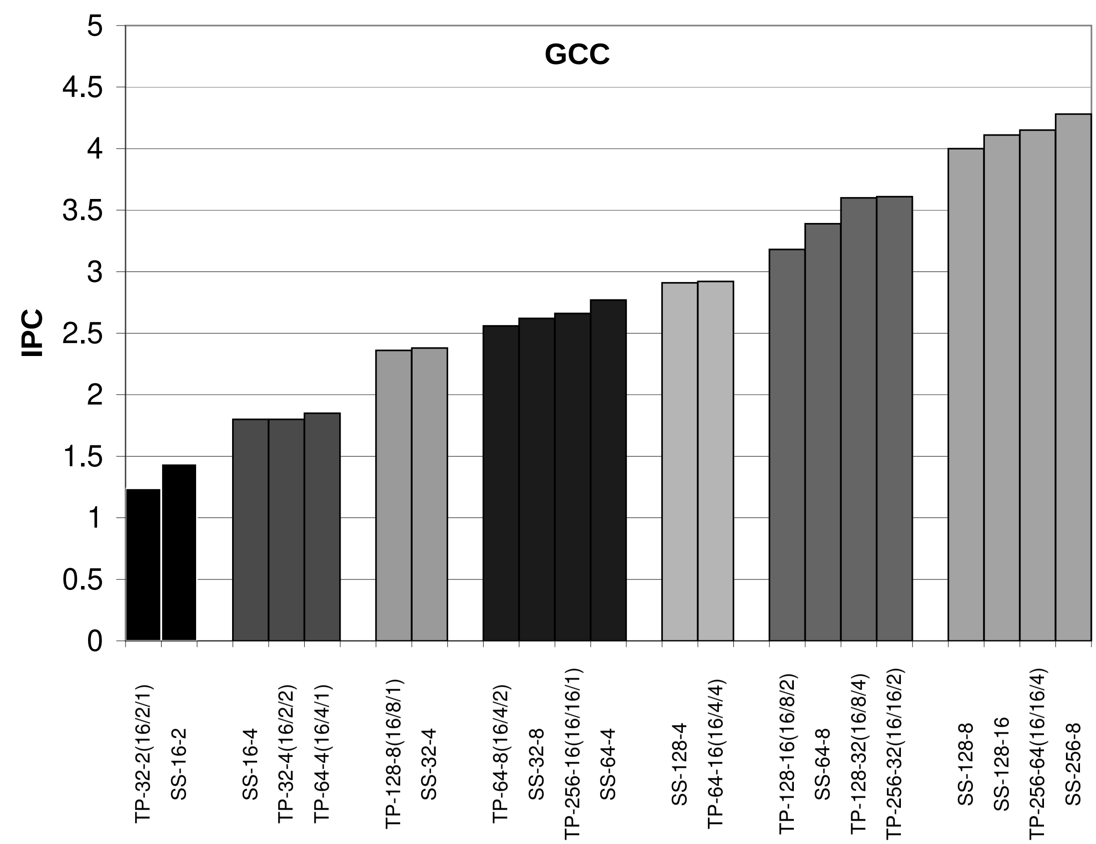
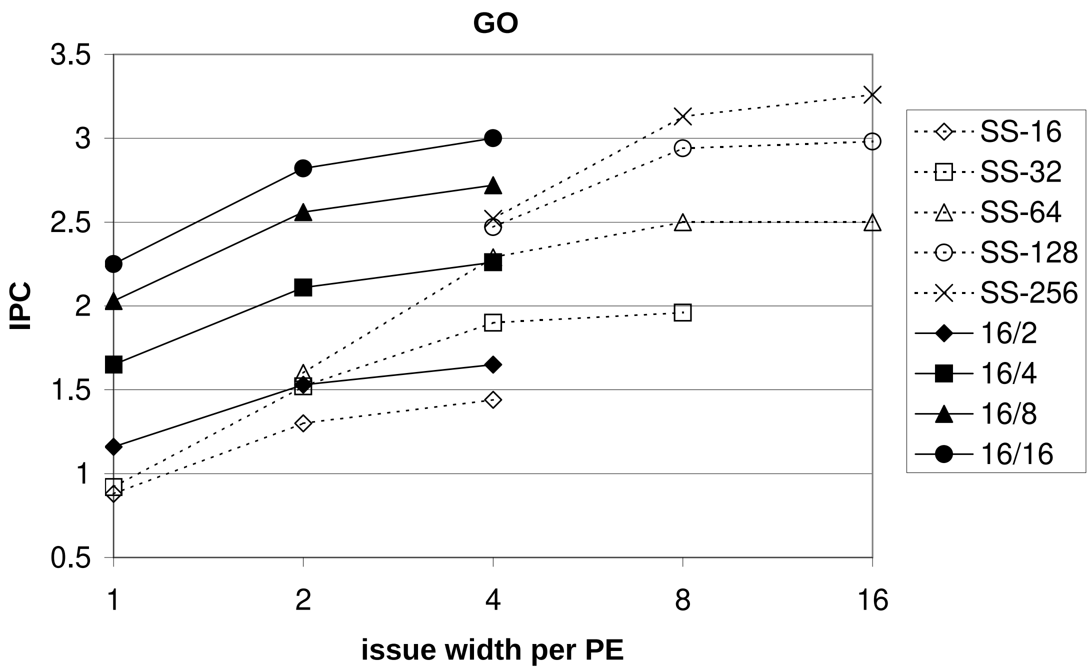


Figure 5-18: Superscalar vs. trace processors with partial bypasses (*gcc*).



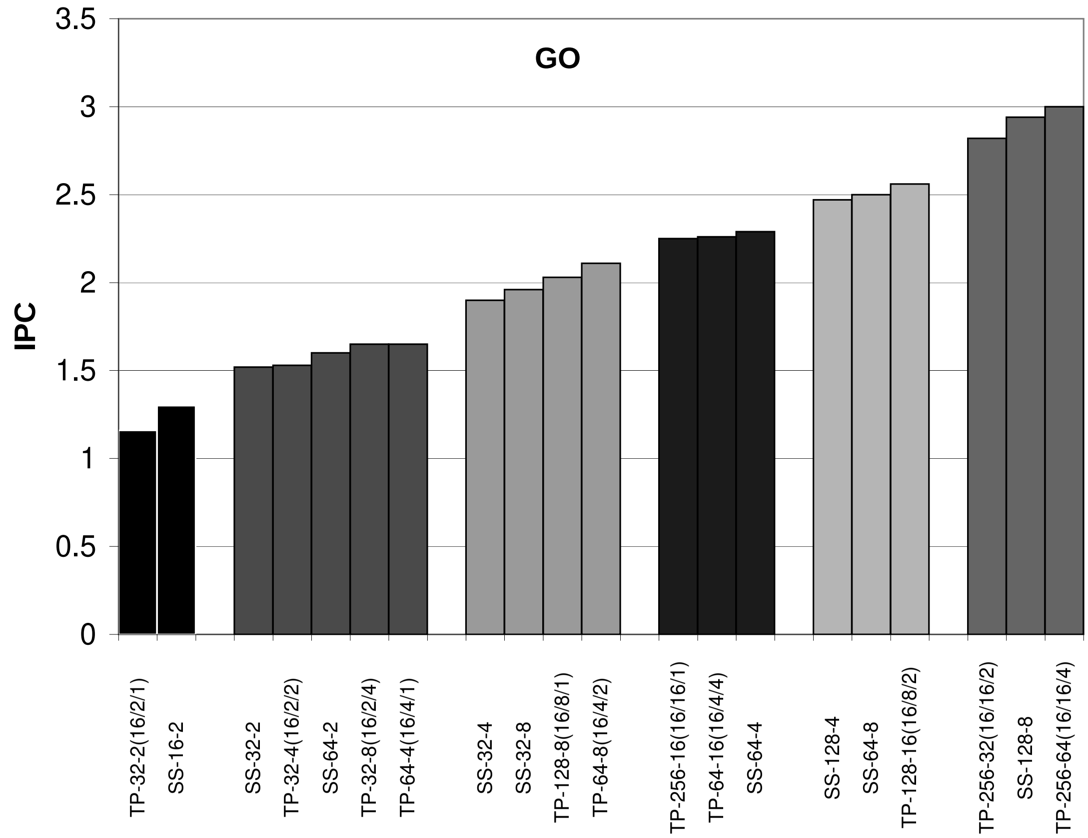


Figure 5-19: Superscalar vs. trace processors with partial bypasses (*go*).

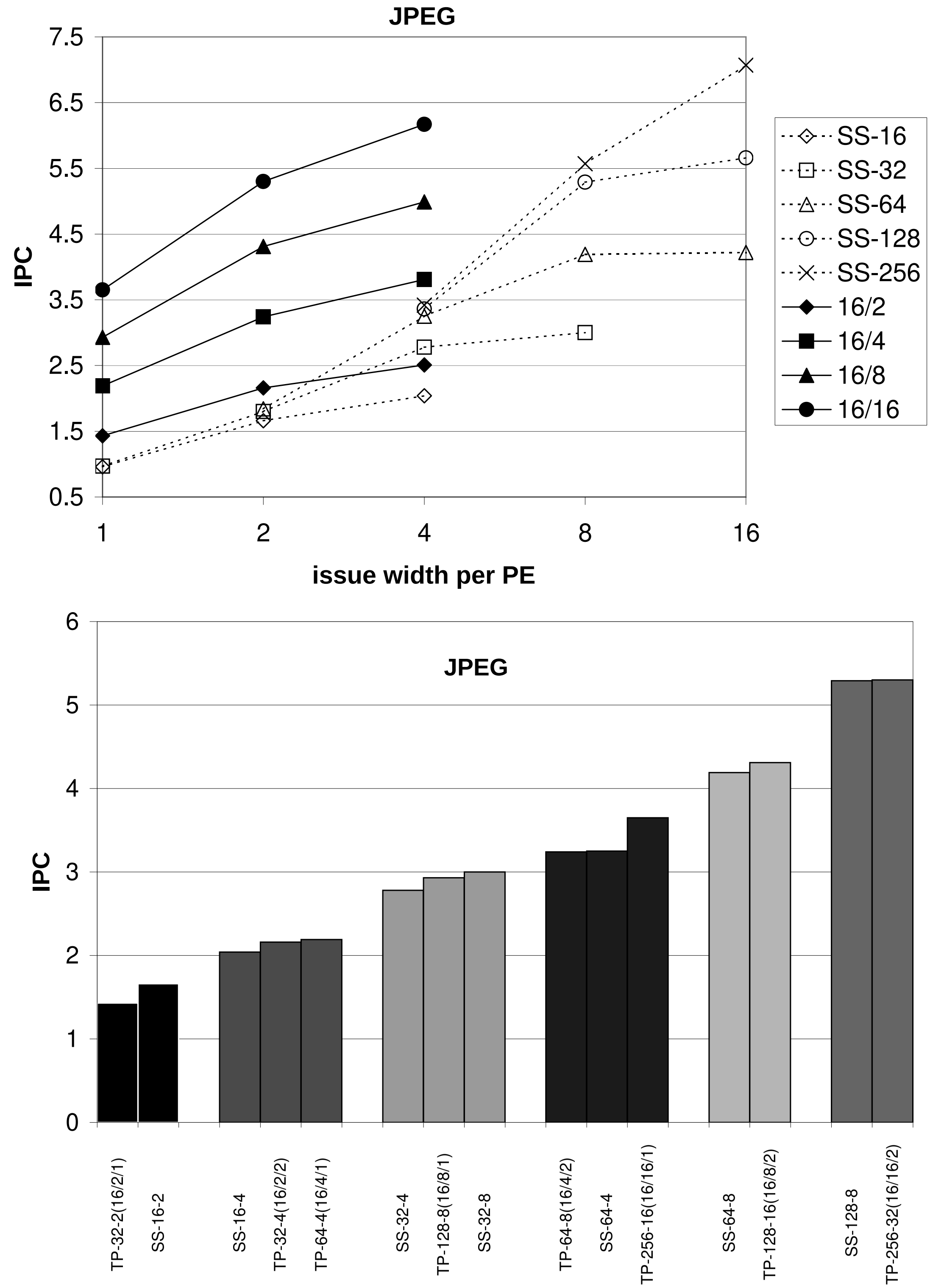


Figure 5-20: Superscalar vs. trace processors with partial bypasses (*jpeg*).

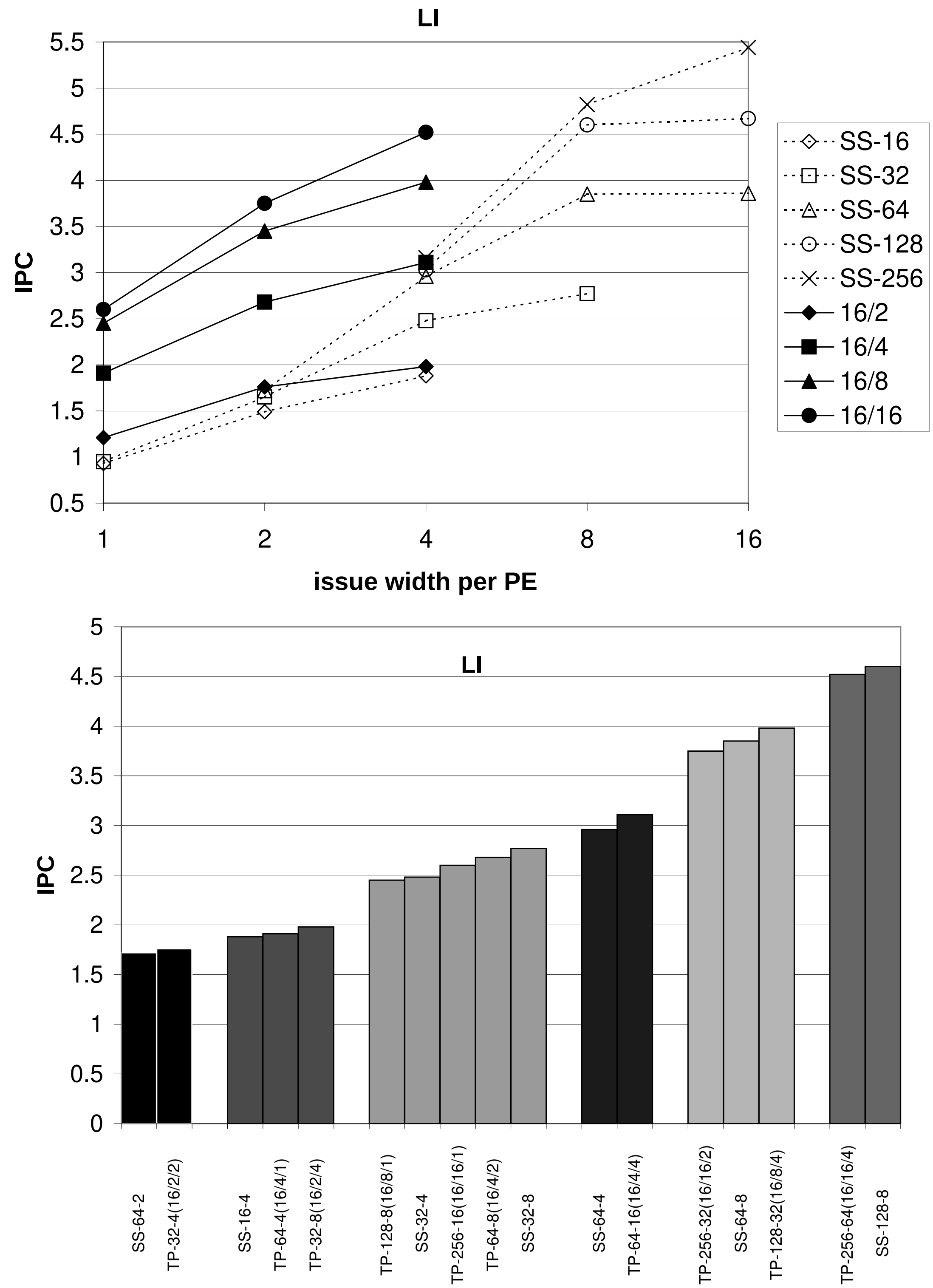
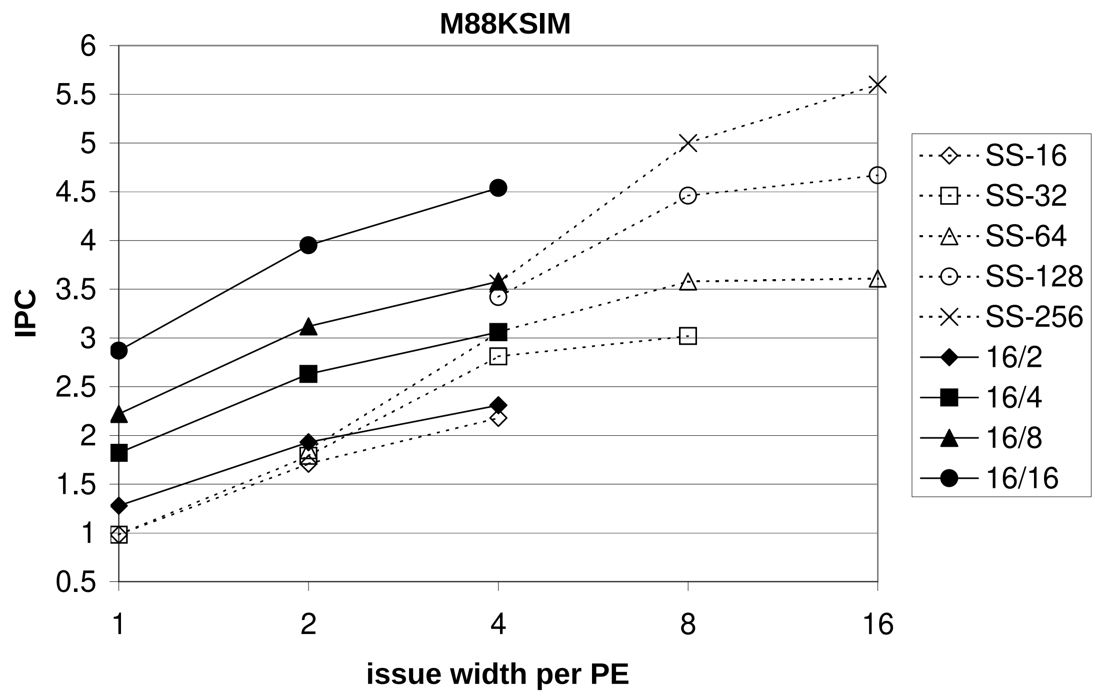


Figure 5-21: Superscalar vs. trace processors with partial bypasses (*li*).



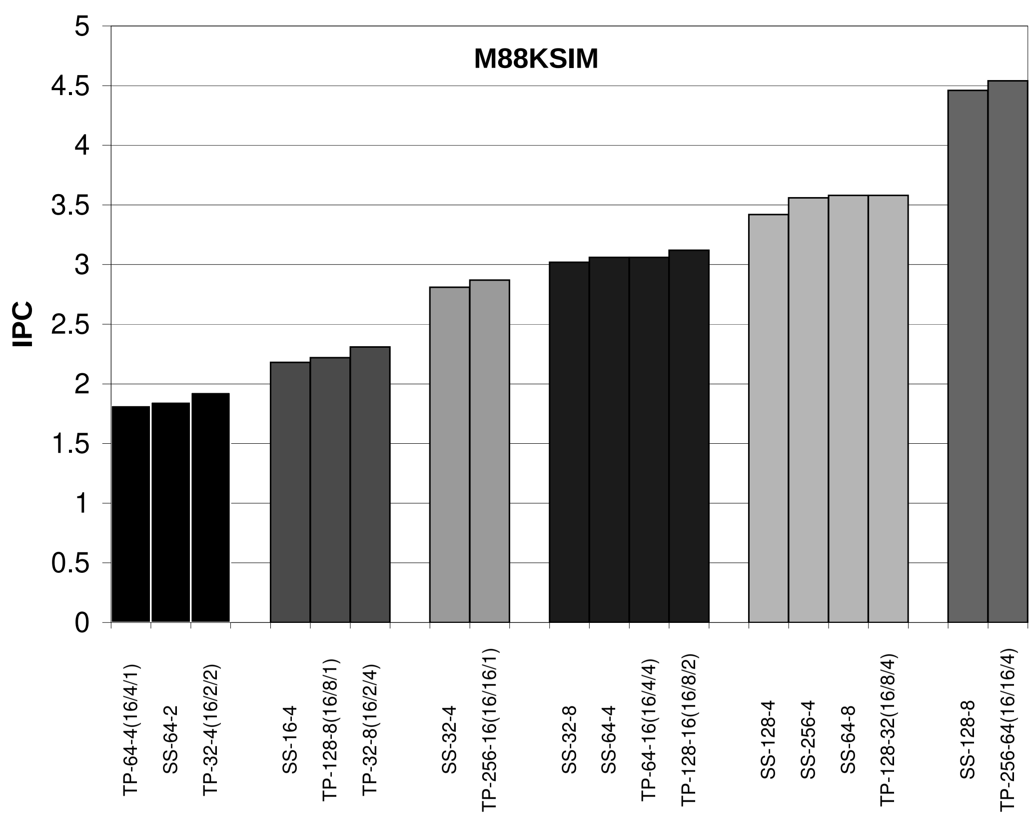
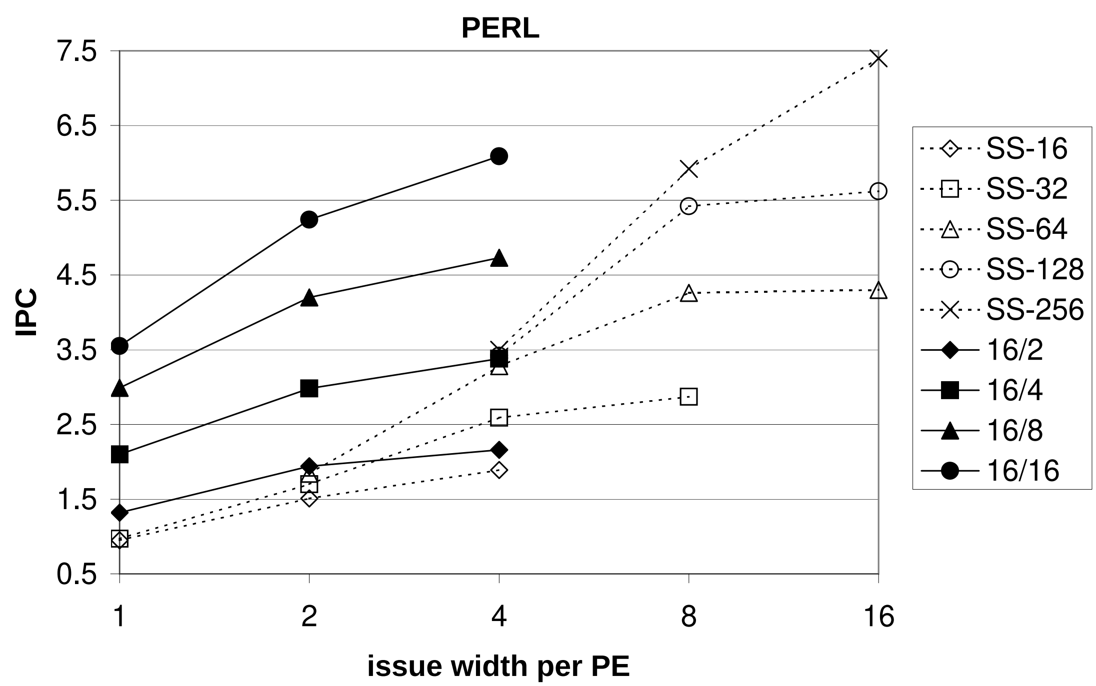


Figure 5-22: Superscalar vs. trace processors with partial bypasses (*m88ksim*).



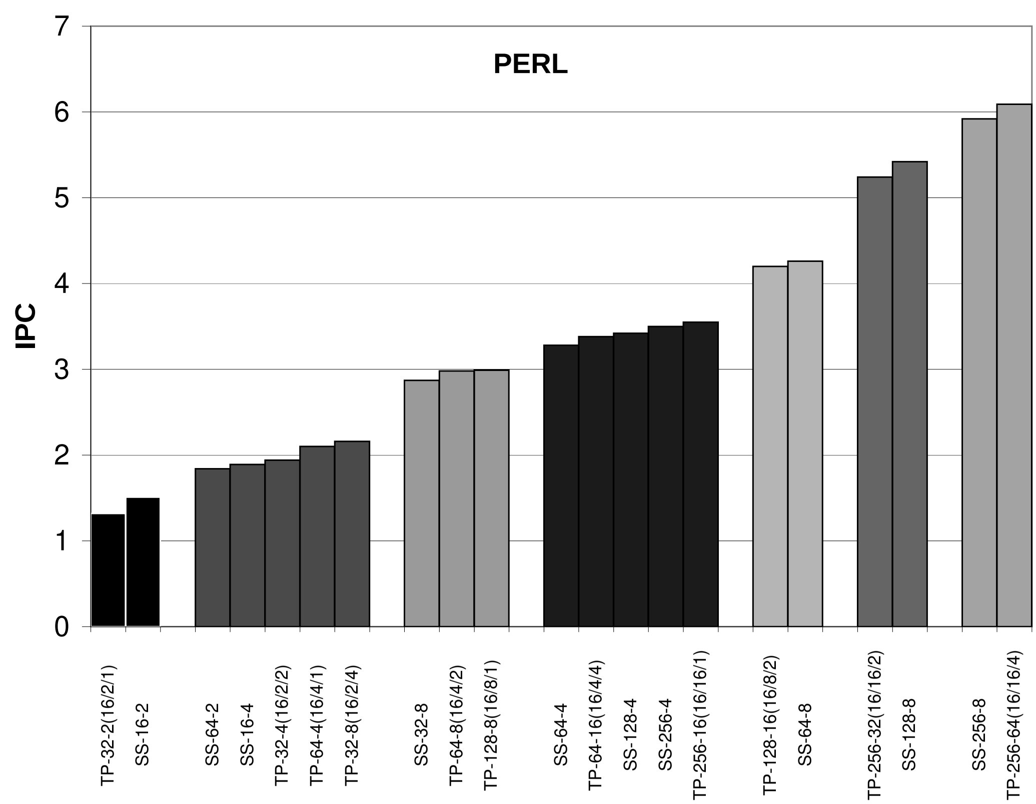


Figure 5-23: Superscalar vs. trace processors with partial bypasses (*perl*).

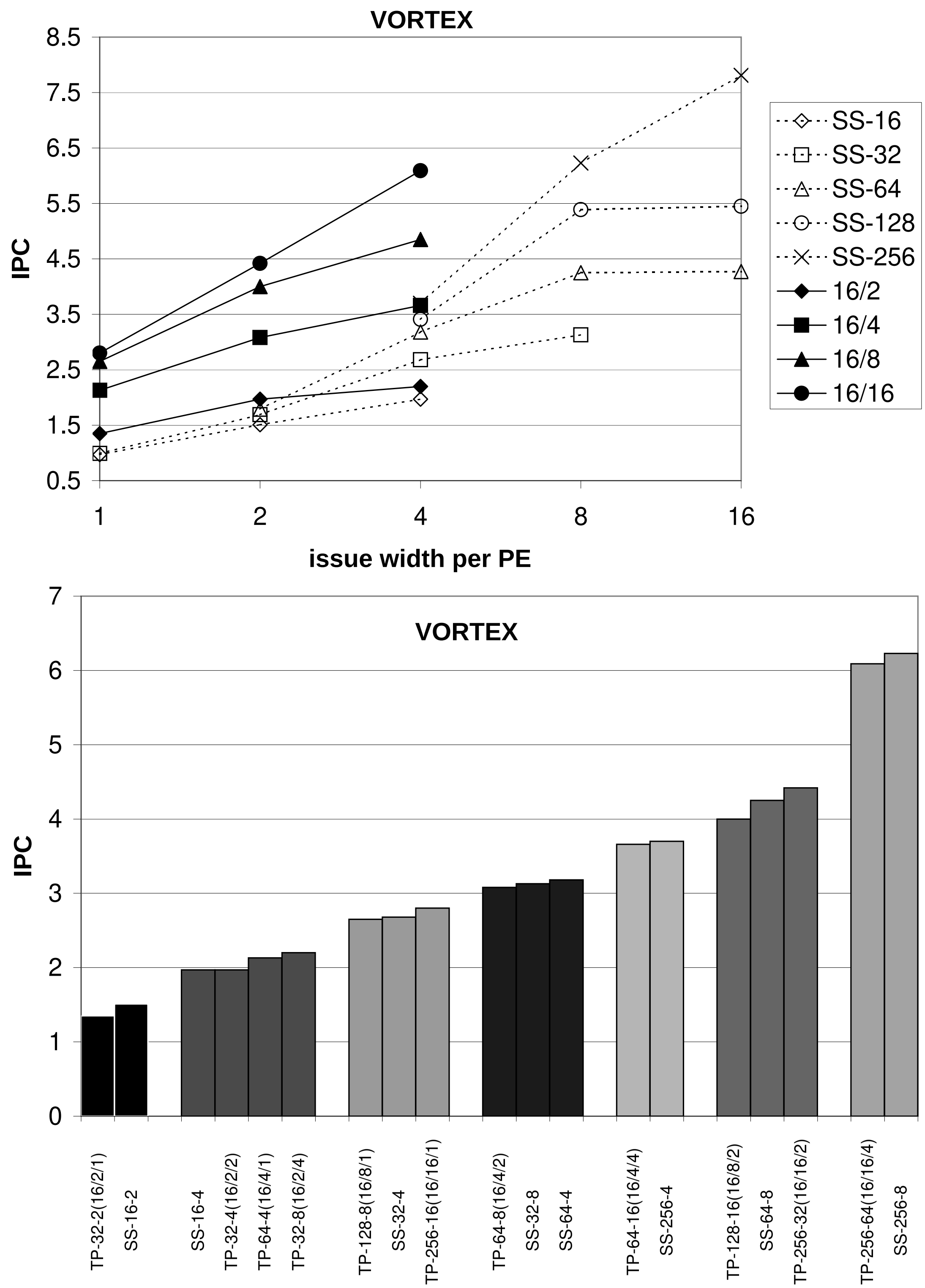


Figure 5-24: Superscalar vs. trace processors with partial bypasses (vortex).

5.2.4 Case studies: comparing overall performance

In this section I estimate the overall performance advantage of trace processors over conventional superscalar processors. First, I analyze circuit complexity to arrive at reasonable cycle time estimates for several trace processor and superscalar processor configurations. Second, these processor configurations are simulated to measure IPC. Overall performance -- *instructions per second* -- is computed as IPC divided by cycle time.

5.2.4.1 Analytical models

The complexity of superscalar processors is quantified in the technical report by Palacharla, Jouppi, and Smith [69]. The report identifies three key areas of complexity -- register renaming, instruction window wakeup and select, and result bypassing -- and derives analytical models for estimating the worst-case delay through each of these circuits. The analytical models are briefly summarized below, and I also describe how the models are applied to trace processors. In several cases, due to the hierarchical organization of trace processors, the inputs to a model change their meaning or the model itself must be modified slightly. I focus on complexity stemming from the instruction window, i.e. rename complexity is not considered here.

- *Wakeup*

  Wakeup involves broadcasting result tags to each instruction in the window. Each instruction compares its operand tags with the result tags to determine if an operand is available. Thus, the primary logic structure is a CAM (content addressable memory) cell. The various delay components of wakeup depend on the dimensions of the cell. For example, the height of the cell determines the overall height of the instruction window, which affects the delay for broadcasting tags.

Both the height and width of the cell depend on the *number of result tags*, NUMTAGS, broadcasted to the window. The overall height of the instruction window depends on the *number of instructions in the window*, WINSIZE. Therefore, wakeup delay is a function of these two variables:

$$T_{wakeup} = f(\text{NUMTAGS}\ ,\ \text{WINSIZE})$$

In [69], it is pointed out that NUMTAGS is simply the issue width of the processor. In trace processors, however, the number of tags is equal to the number of local result buses plus the number of global result buses.

I do not reproduce the full delay equation here because it is not actually required -- the technical report tabulates results over a range of window sizes and issue widths. Relevant results are reproduced in Table 5-4. For trace processors, the PE window size and sum of local and global tags are used to lookup the corresponding delay in the table.

It turns out that wakeup delay is much more sensitive to NUMTAGS than WINSIZE [69]. WINSIZE affects only one of three major wakeup delay components, whereas NUMTAGS affects all three. The trace processor does a good job of reducing WINSIZE, in fact, WINSIZE remains constant as more PEs are added. Unfortunately, NUMTAGS is potentially problematic despite the hierarchical result bus model.

- *Select*

The select logic receives individual request signals from all instructions in the window and outputs as many grant signals. It is organized as a tree of arbiter cells. Each cell ORs together four request signals into a single request signal to be propagated to the next level of the tree, until ultimately the root cell is reached. From the root cell, grant signals are propagated and expanded back up the tree.

The arbitration scheme works for selecting a single instruction to execute on a single functional unit. If there are multiple functional units of the same type, there are at least two options. The select logic modules for each functional unit may be serially cascaded; request signals are gated by not-grant signals before routing the updated requests into successive select logic modules. Alternatively, the arbiter cell may be extended so that request and grant signals indicate the number of resources requested or granted. As pointed out in [69], the extended arbiter cell becomes quite complex even for just two functional units, and serially cascading the much simpler select logic modules is likely faster.

Select delay is a function of $\log_4(\text{WINSIZE})$ (due to the tree structure), and if there are N copies of the most common functional unit type, the delay is multiplied by a factor of N (due to cascading).

$$T_{select} = N * f(\log_4(\text{WINSIZE}))$$

The full delay equation is not reproduced here because it is not actually required -- as with wakeup delay, the technical report tabulates select delay over a range of window sizes. Relevant results are reproduced in Table 5-5 for a single functional unit. I merely apply the multiplicative factor N as needed to account for more functional units.

- *Result bypassing*

By far, the most significant delay component of the result bypass logic is the bus *wire*. Result bus wires stretch linearly above all of the functional units and the register file. In this way, results can be written to or read from the bus by any functional unit or the register file.

The delay of a result bypass wire of length L is given by:

$$T_{bypass} = 0.5 * L^2 * R_{metal} * C_{metal}$$

R_{metal} and C_{metal} are resistance and capacitance per unit length of wire, and their values are given in Table 5-6. The length L is simply the sum of the heights of functional units and the register file. The lambda-based (technology independent) functional unit heights reported in [69] are listed in Table 5-7. The height of a superscalar register file is given by:

$$RF_{height} = NPREG * (cellheight + portheight * (3 * IW))$$

NPREG is the total number of physical registers, *cellheight* is the height of a register file cell in isolation, and *portheight* is the height added to a cell due to a single port.

Lambda-based values for *cellheight* and *portheight* are given in Table 5-8. The factor (3 * IW) is the total number of ports for a IW-wide machine assuming 2 read ports and 1 write port is required for each instruction issued per cycle.

The equation for RF_{height} must be modified for a trace processor due to the hierarchical register file, and is given by:

$$\begin{aligned} RF_{height} = {} & GRF * (\mathit{cellheight} + \mathit{portheight} * (2 * IW_{PE} + GRB)) + \\ & LRF * (\mathit{cellheight} + \mathit{portheight} * (3 * IW_{PE})) \end{aligned}$$

The above equation has two components. The first is for the global register file and the second is for the local register file. GRF and LRF are the sizes of the global and local register files, respectively, IW_{PE} is the issue width per PE, and GRB is the number of global result buses. The factor ($2 * IW_{PE}$ + GRB) is the number of global register file ports. There is a one-to-one correspondence between global result buses and global register file write ports, consequently, there are GRB write ports. Because each PE has a copy of the global register file, the number of read ports is equal to the issue width of the PE times two for each instruction operand. The local register file component of the equation is a straightforward translation of the superscalar register file equation.

Table 5-4: Wakeup delays for 0.18μm technology. Obtained from [69].

Window Size	Wakeup Delay (ps)
Issue Width = 2	
16	148.1
32	158.6
64	180.5
Issue Width = 4	
16	190.6
32	206.3
64	241.3
Issue Width = 8	
16	258.3
32	285.1
64	352.3

Table 5-5: Select delays for 0.18μm technology, assuming a single functional unit. Obtained from [69].

Window Size	Select Delay (ps)
16	250.4
32	371.7
64	373.0
128	502.1

Table 5-6: Technology constants. Obtained from [69].

Technology	λ	R_{metal} (Ω/μm)	C_{metal} (fF/μm)
0.18μm	0.09μm	0.09	1.22

Table 5-7: Functional unit heights. Obtained from [69].

Functional unit	Height (λ)	Description
Complete ALU (ALU_{full})	3200	Comprises adder, shifter, and logic unit
Simple ALU (ALU_{simple})	1700	Comprises adder and logic unit
Address Generation Unit (*AGEN*)	1400	Comprises adder for effective address calculation

Table 5-8: Register file circuit parameters. Obtained from [69].

Parameter	Value
cellheight	40λ
portheight	10λ

5.2.4.2 Quantifying complexity

In my case study, the critical delays of one trace processor and six superscalar processors are measured. `TP-128-16(16/8/2)` is an interesting design point because it represents a reasonably large trace processor with high performance and low PE complexity. The data in Table 5-3 indicates `TP-128-16(16/8/2)` and `SS-64-8` perform comparably for most of the benchmarks. For various other benchmarks, `SS-64-4` or `SS-128-4` give similar performance. Therefore, I consider superscalar processors with 64 and 128 instruction windows and both 4-way and 8-way issue.

Two smaller superscalar processors, `SS-16-2` and `SS-32-2`, are also considered. `SS-16-2` is the basic building block of the trace processor and should provide an interesting comparison. I expect its cycle time to be shorter than that of the trace processor because global circuitry is absent. The comparison will demonstrate the net performance gain when multiple small-sized superscalar processors are connected together in a trace processor.

Table 5-9 shows the parameters used in the wakeup, select, and bypass delay models for each of the processor configurations. The wakeup model requires two parameters, NUMTAGS and WINSIZE. The number of tags broadcast to the PE window is the total number of local and global result tag buses. There are 2 local result buses in a 2-way issue

PE. Data in Section 5.4.1 indicates 6 global result buses perform as well as unlimited result buses for `TP-128-16(16/8/2)`. Thus, NUMTAGS equals 8 in the trace processor. I point out that 4 global result buses perform almost as well as 6. Although I could parameterize the wakeup model using 4 or 5 global result buses, that would make for 6 or 7 NUMTAGS, and there are no delays in Table 5-4 corresponding to these datapoints. For the superscalar processors, NUMTAGS is equal to the total issue width.

The select model requires two parameters, WINSIZE and N. WINSIZE is equal to PE window size in the case of the trace processor, or total window size in the case of superscalar processors. N is based on the number and type of functional units. Although IPC is measured using fully symmetric functional units, the complexity analysis uses a reasonable mix of functional units described in [69] and reproduced in Table 5-10. I assume 1) ALU operations are routed to either the ALU_{full} or ALU_{simple} functional units, and 2) if there are no separate AGEN units, then address computations are performed by both ALU_{full} and ALU_{simple} functional units, otherwise address computations are performed only by *AGEN*. Based on these assumptions, N is computed is follows.

- 2-way issue: N = 2 because arbitration must be performed for the two ALUs together.
- 4-way issue: N = 2 because arbitration involves a maximum of two functional units, i.e. either two ALUs or two AGENs.
- 8-way issue: N = 4 because arbitration involves a maximum of four functional units, i.e. either four ALUs or four AGENs.

The bypass delay model is parameterized by four numbers in the trace processor, GRF (global register file size), LRF (local register file size), IW_{PE} (PE issue width), and GRB (number of write ports to the global register file, i.e. number of global result data buses). Measurements in Section 5.4.3 indicate a global register file of 80 registers performs as well as an unconstrained register file, for `TP-128-16(16/8/2)`. The average number of local values per trace (Table 4-5) suggests a local register file of 8 registers is adequate. And, as stated previously, there are 6 global result data buses. For superscalar processors, bypass delay is parameterized by physical register file size and issue bandwidth. NPREGS is equal to 80 for `SS-64` and 120 for `SS-128`, in accordance with [69]. I assume 40 physical registers for `SS-16-2` (32 architected + 8 speculative) and 60 for `SS-32-2`.

Table 5-9: Parameter values for wakeup, select, and bypass models.

Processor Configuration		*Select*		*Bypass*			
	Wakeup						
	NUMTAGS	WINSIZE	N	GRF / NPREG	LRF	IW / IW_{PE}	GRB
TP-128-16 (16/8/2)	8	16	2	80	8	2	6
SS-16-2	2	16	2	40	-	2	-
SS-32-2	2	32	2	60	-	2	-
SS-64-4	4	64	2	80	-	4	-
SS-64-8	8	64	4	80	-	8	-
SS-128-4	4	128	2	120	-	4	-
SS-128-8	8	128	4	120	-	8	-

Table 5-10: Functional unit mix for various processor/PE issue widths. Mixes for 4-way and 8-way issue are obtained from [69].

Issue width	Functional unit mix
2	1 ALU_{full} , 1 ALU_{simple}
4	1 ALU_{full} , 1 ALU_{simple} , 2 $AGEN$
8	2 ALU_{full} , 2 ALU_{simple} , 4 $AGEN$

Finally, computed delays and estimated cycle times are shown in Table 5-11. Wakeup delay is given in the first column. Wakeup delays for windows larger than 64 instructions are unavailable because they are not investigated in [69]; the authors suggest larger superscalar windows should be formed by banking. Along those lines, wakeup delays for `SS-128` configurations are estimated using the delays of corresponding `SS-64` configurations and adding about 10% for the overhead of banking (hence the ~ symbol preceding the `SS-128` wakeup delays).

Wakeup delay for the trace processor is 25% faster than wakeup delay for `SS-64-8`. Since both processors broadcast 8 tags, this improvement is due entirely to the small PE window. `SS-64-4` has slightly faster wakeup logic than the trace processor because NUMTAGS affects delay more than WINSIZE. In spite of a small PE window, wakeup in the trace processor is still a potential problem because the number of tags is not reduced as much as one would like. On the other hand, as suggested earlier, only 4 or 5 global tags may be implemented to get nearly the same IPC performance and an even greater reduction in wakeup delay.

The effect of NUMTAGS is evident by comparing the trace processor and `SS-16-2`. The window size is 16 in both cases, but `SS-16-2` broadcasts only 2 tags. `SS-16-2` wakeup delay is consequently 40% faster.

Select delay is shown in the second column of Table 5-11. Interestingly, select logic is the larger component of total issue delay (third column, wakeup plus select delay). Select delay increases substantially with both window size and issue bandwidth. It is particularly sensitive to issue bandwidth because arbitration complexity increases linearly with the number of functional units. The modest, dedicated issue bandwidth within PEs, coupled with the small PE window, substantially reduces arbitration complexity. Essentially, select logic is fully distributed in the trace processor. Select delay is reduced by more than 60% with respect to `SS-64-8` and 50% with respect to `SS-128-4`.

Local bypass wire lengths and resulting delays are shown in the 6^{th} and 7^{th} columns. The 8-way issue superscalar processors have substantially longer bypass delays because of the quadratic dependence on bypass wire length.

The trace processor has relatively short local bypass wires for two reasons. Firstly, a PE has only two functional units. Secondly, the global register file is smaller than the register file of an equivalent superscalar processor; there are fewer ports and fewer registers due to the hierarchical register model. Note that the local register file is also included in the wire length computation. With respect to `SS-64-8`, the length of wire running over functional units and register files is reduced by about 70% and 50%, respectively, for an overall reduction in bypass delay of 80%.

The bypass delay of the smaller superscalar processor `SS-64-4` is more comparable to that of the trace processor. Substantially fewer ports are required for 4-way issue than 8-way issue, as a result, the length of wire running over the `SS-64-4` register file and the trace processor's local plus global register files is comparable. The bypass delay of `SS-16-2` is fairly negligible and 75% faster than the trace processor's local bypass delay. The length of wire running over the `SS-16-2` register file is 66% shorter because it has half the number of physical registers and four fewer write ports than the trace processor's global register file.

An analytical model for global bypass delay is beyond the scope of this study. The lengths of global result wires depend on the overall trace processor floorplan. As shown in the second-to-last column of Table 5-11, global bypass delay is estimated to be about 1000 ps based on the bypass delay of the comparably-sized `SS-128-8` configuration.

I am now ready to make cycle time estimations for each of the configurations.

- `TP-128-16(16/8/2)`

  A range of cycle times is given for the trace processor. If global bypass wires set the critical path at 1000 ps, then the cycle time is 1 ns. Suppose, however, that global bypass can meet the timing constraints set by the issue logic (wakeup and select). Then the cycle time is about 0.8 ns. The last column in Table 5-11 shows the range of cycle times for the trace processor: 0.8 ns to 1.0 ns. If wakeup and select set the cycle time at 0.8 ns, then the implied latency for performing an add instruction or similar ALU operation is just under 700 ps, because local bypass takes 130 ps.

These estimates, including the 700 ps ALU operation, are consistent with the *Semiconductor Roadmap's* projections for 0.18μm technology [90] and projections for forthcoming 1^+ GHz processors.

- Superscalar processors

Wakeup and select delays dictate the cycle time estimates for all of the superscalar processors. Cycle time estimates based on wakeup and select delay are shown in the last column of Table 5-11 (the first number listed). The 8-way issue processors have considerably longer cycle times than 4-way issue processors: 1.9 ns to 2.4 ns for 8-way versus 1.0 ns to 1.3 ns for 4-way, for the two window sizes. Among the medium-size superscalar processors, the cycle times of all but `SS-64-4` are considerably longer than the trace processor cycle time.

The cycle time estimate for `SS-16-2` is noticeably better than the trace processor's upper bound estimate. This is due to the global bypass delay estimate. But the trace processor's lower bound estimate is only slightly worse than `SS-16-2` cycle time.

For superscalar processors, a second cycle time is indicated within parentheses in the last column. Suppose one could improve the select logic and, instead, bypass delay were the critical path. The second cycle time estimate reflects the combined ALU latency (700 ps, as described above) and bypass delay. The cycle time does not change much for `SS-64-4`, `SS-32-2`, or `SS-16-2` because their delays are well balanced. The cycle times of other superscalar processors decrease by 25% to 30%. Nevertheless, bypass delay is substantial -- e.g. for `SS-64-8`, bypass delay nearly equals the latency of an ALU operation.

Table 5-11: Delay computations and cycle time estimates.

Processor Config.	Wakeup delay (ps)	Select delay (ps)	Wakeup + Select delay (ps)	Local Bypass				Global Bypass delay (ps)	cycle time (ns)
				Σ FU_{height} (λ)	RF_{height} (λ)	wire length (λ)	Delay (ps)		
TP-128-16 (16/8/2)	260	500	760	4900	12000	16900	130	< 1000	0.8 - 1.0
SS-16-2	150	500	650	4900	4000	8900	35	-	0.7 (0.7)
SS-32-2	160	740	900	4900	6000	10900	53	-	0.9 (0.8)
SS-64-4	240	750	990	7700	12800	20500	190	-	1.0 (0.9)
SS-64-8	350	1500	1850	15400	22400	37800	640	-	1.9 (1.3)
SS-128-4	~270	1000	1300	7700	19200	26900	320	-	1.3 (1.0)
SS-128-8	~390	2000	2400	15400	33600	49000	1100	-	2.4 (1.8)

5.2.4.3 Overall performance

The graphs in Figures 5-25 through 5-31 show the *overall performance improvement* of `TP-128-16(16/8/2)` relative to the six superscalar configurations as *relative cycle time* decreases (each graph has six curves, one per superscalar configuration). More precisely:

1. The x-axis is the reduction in cycle time of the trace processor with respect to the superscalar processor. Note: *T* is cycle time and *x* is the cycle time reduction. The *SS* and *TP* subscripts denote superscalar and trace processors.

$$x = \frac{T_{SS} - T_{TP}}{T_{SS}}$$

2. The y-axis is the overall performance improvement of the trace processor with respect to a superscalar processor and is computed as follows. Note: *IPC* is instructions per cycle, *T* is cycle time, and *IPS* is instructions per second (i.e. overall performance).

$$y = \frac{IPS_{TP} - IPS_{SS}}{IPS_{SS}} = \frac{\frac{IPC_{TP}}{T_{TP}} - \frac{IPC_{SS}}{T_{SS}}}{\frac{IPC_{SS}}{T_{SS}}} = \frac{\frac{IPC_{TP}}{(1-x)T_{SS}} - \frac{IPC_{SS}}{T_{SS}}}{\frac{IPC_{SS}}{T_{SS}}} = \frac{\left(\frac{\mathbf{IPC_{TP}}}{\mathbf{1-x}}\right) - \mathbf{IPC_{SS}}}{\mathbf{IPC_{SS}}}$$

Referring to the graphs in Figures 5-25 through 5-31, except for *li* and *m88ksim*, the curve labeled `SS-64-8` crosses near the origin. This reconfirms that the trace processor and `SS-64-8` have similar IPC. But as the clock period of the trace processor decreases with respect to the clock period of `SS-64-8`, the trace processor gives better overall performance, i.e. more instructions executed per second.

Consider the graph for *gcc*, which is fairly representative of the other benchmarks. Because the trace processor has slightly lower IPC than `SS-64-8`, a 5% shorter clock period is needed to break even in terms of overall performance. From Table 5-11, the trace processor clock period is from 47% to 58% shorter than the `SS-64-8` clock period (0.8 ns to 1.0 ns for the trace processor versus 1.9 ns for the superscalar processor). These two endpoints are plotted directly on the `SS-64-8` curve and connected with a visible arc. From the arc, I observe that the trace processor performs from 80% to 120% better than `SS-64-8` based on cycle time estimates and IPC measurements.

From the same graph, I observe that the trace processor performs about 15% better than `SS−64−4` given the same clock period. The arc on this curve indicates that the clock period of the trace processor is from 0% to 20% shorter than the clock period of SS-64-4, and this corresponds to an overall performance improvement of up to 45%. About 1/3 of the overall improvement is due to IPC and 2/3 to clock rate.

The trace processor's clock period is from 15% to 40% *longer* than the clock period of `SS−16−2`. But the trace processor has significantly higher IPC and the net result is an overall performance improvement. For example, *gcc* shows an overall improvement of about 50% to 90%. Evidently, it is beneficial to connect multiple, small superscalar building blocks into a trace processor. The incremental clock period overhead is absorbed and the increase in IPC provides a substantial payoff.

The three configurations discussed so far (`SS−16−2`, `SS−64−4`, and `SS−64−8`) form a complete picture. The trace processor performs better than both `SS−16−2` and `SS−64−8`, but for different reasons. On the one hand, `SS−16−2` has a fast cycle time but its IPC is too low to be competitive; on the other hand, `SS−64−8` has reasonably high IPC but its cycle time is too slow. `SS−64−4` is closest to the trace processor both in terms of cycle time and IPC. Evidently, `SS−64−4` is the best superscalar design point in terms of overall performance.

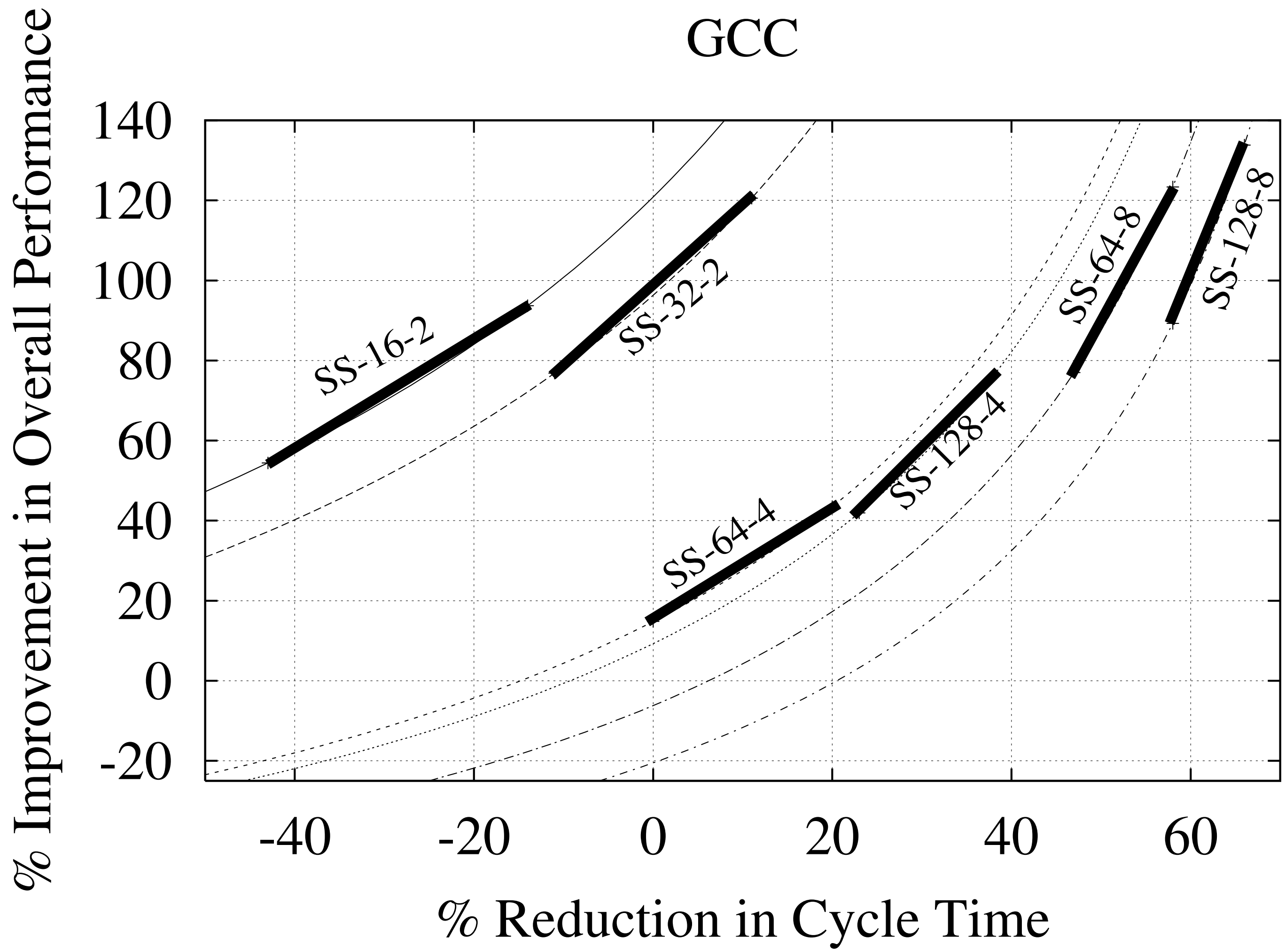


Figure 5-25: TP-128-16(16/8/2) vs. six superscalar processors (*gcc*).

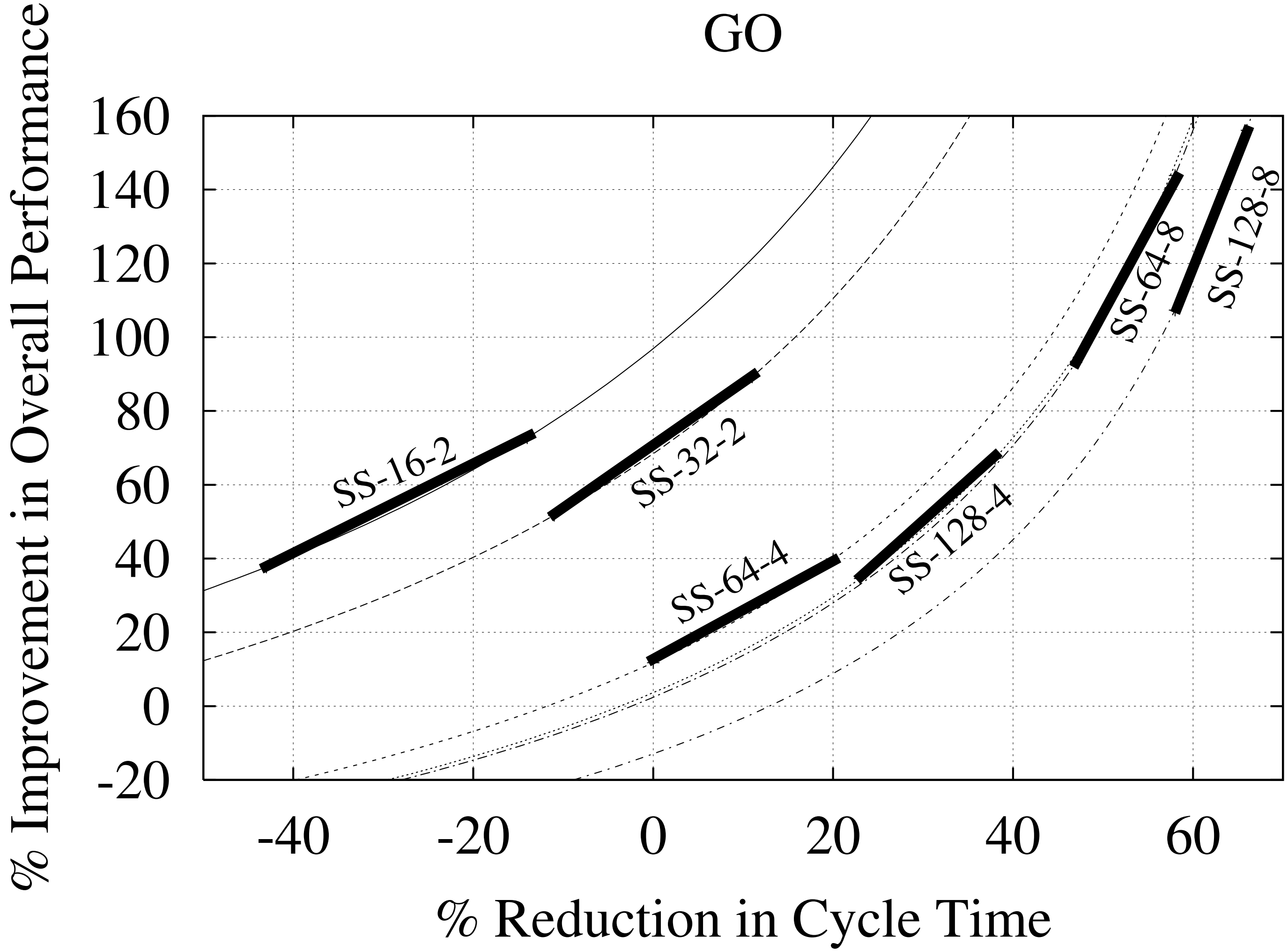


Figure 5-26: TP-128-16(16/8/2) vs. six superscalar processors (go).

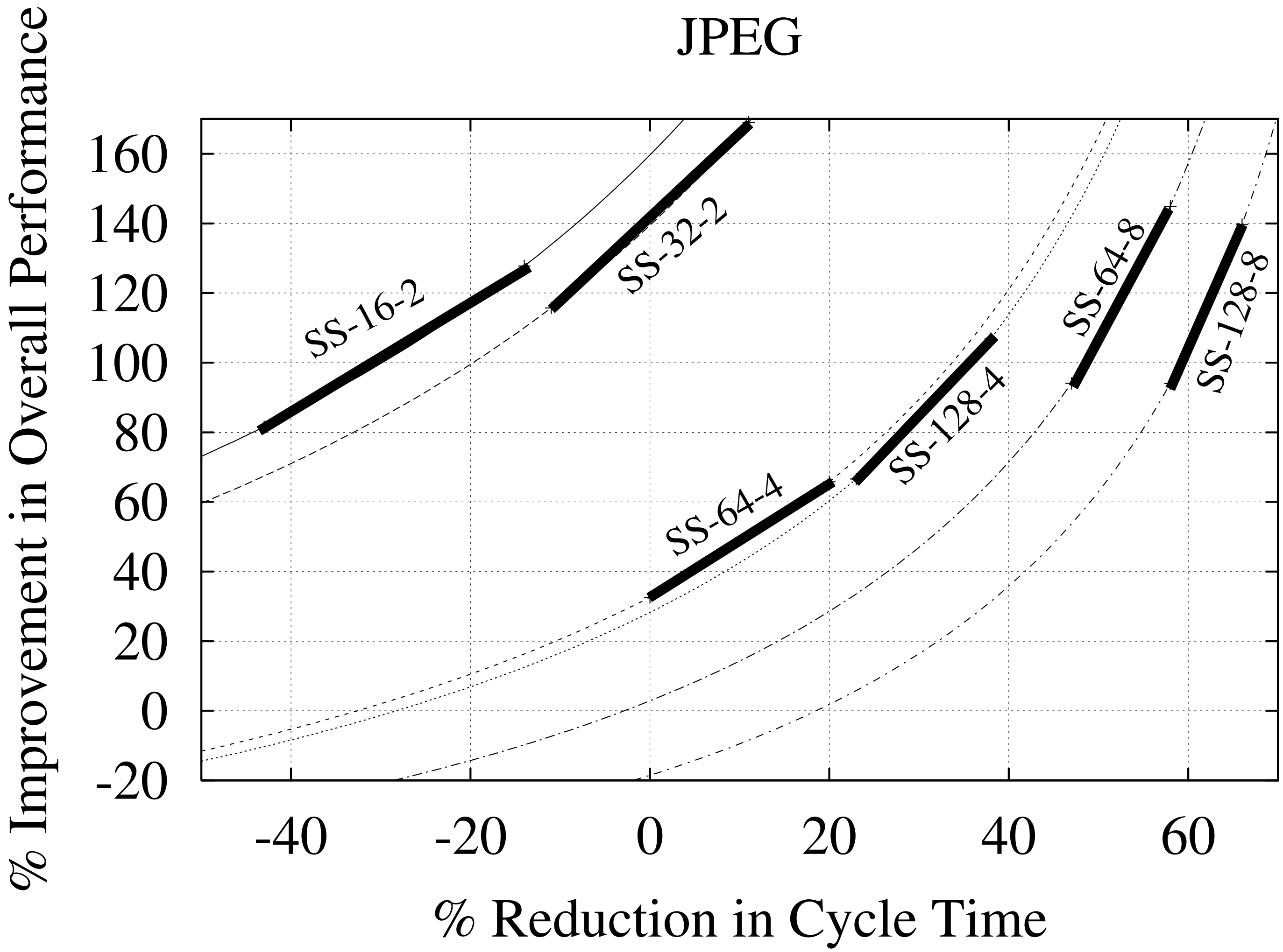


Figure 5-27: TP-128-16(16/8/2) vs. six superscalar processors (*jpeg*).

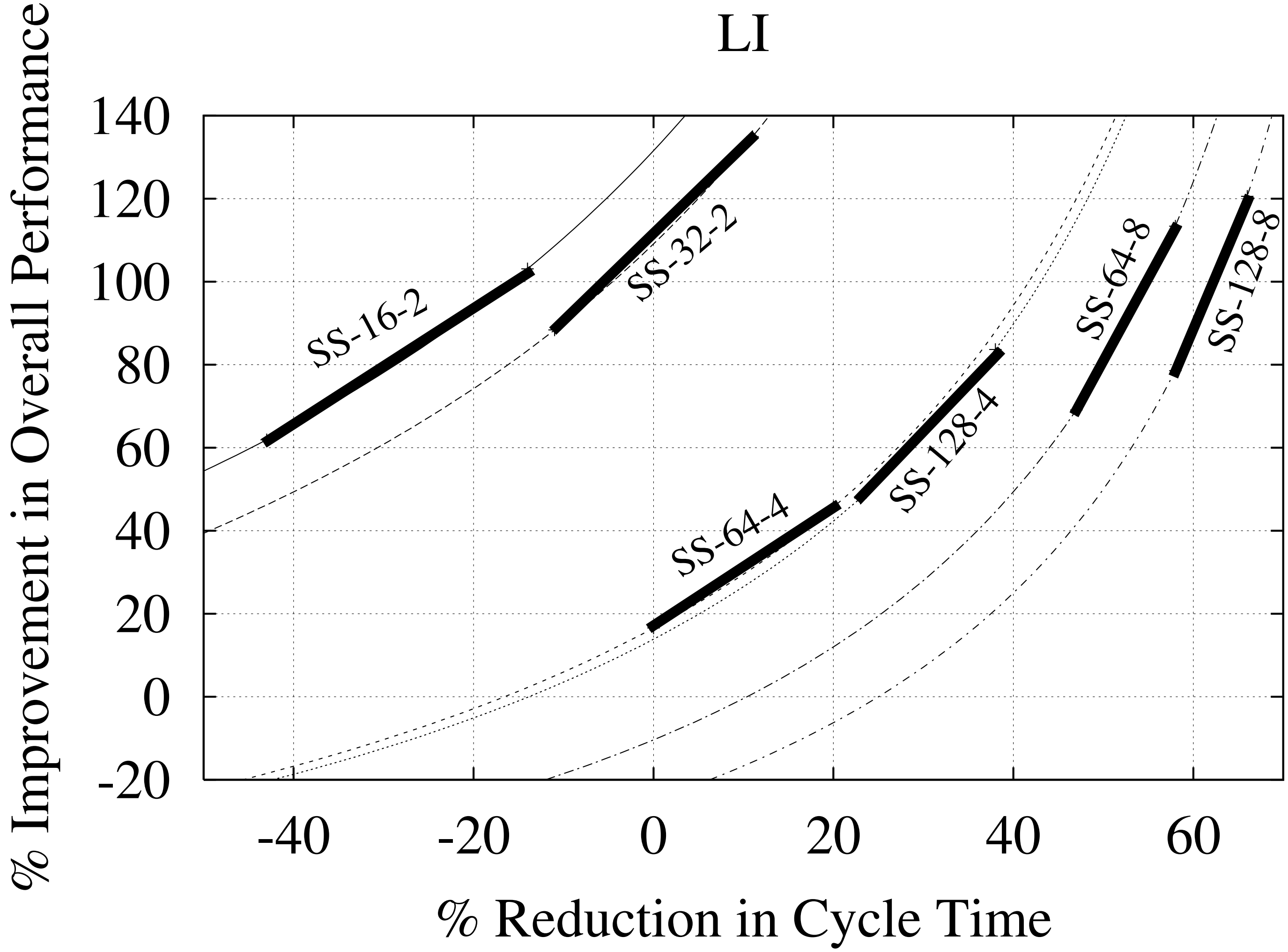


Figure 5-28: TP-128-16(16/8/2) vs. six superscalar processors (*li*).

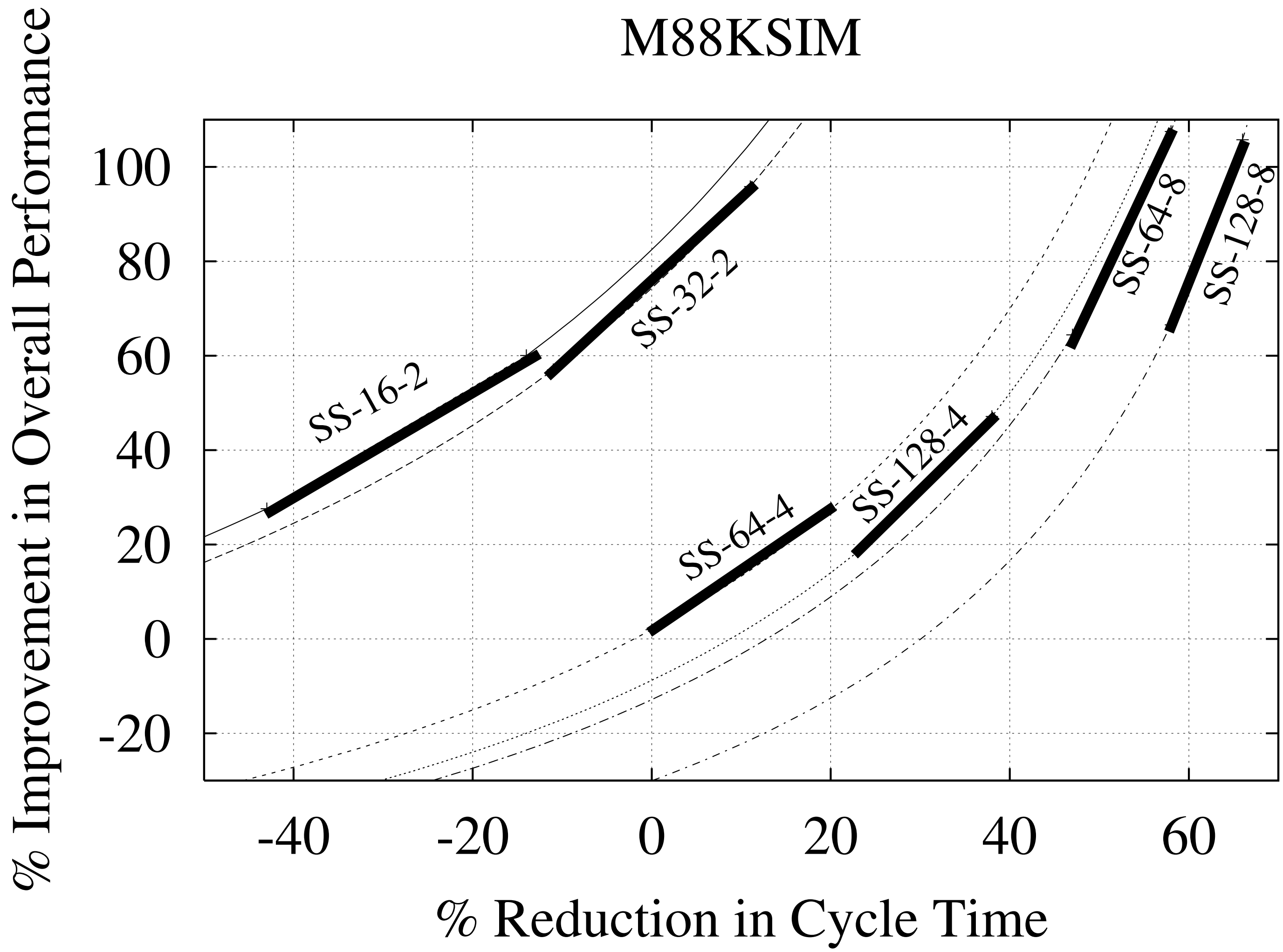


Figure 5-29: TP-128-16(16/8/2) vs. six superscalar processors (*m88k*).

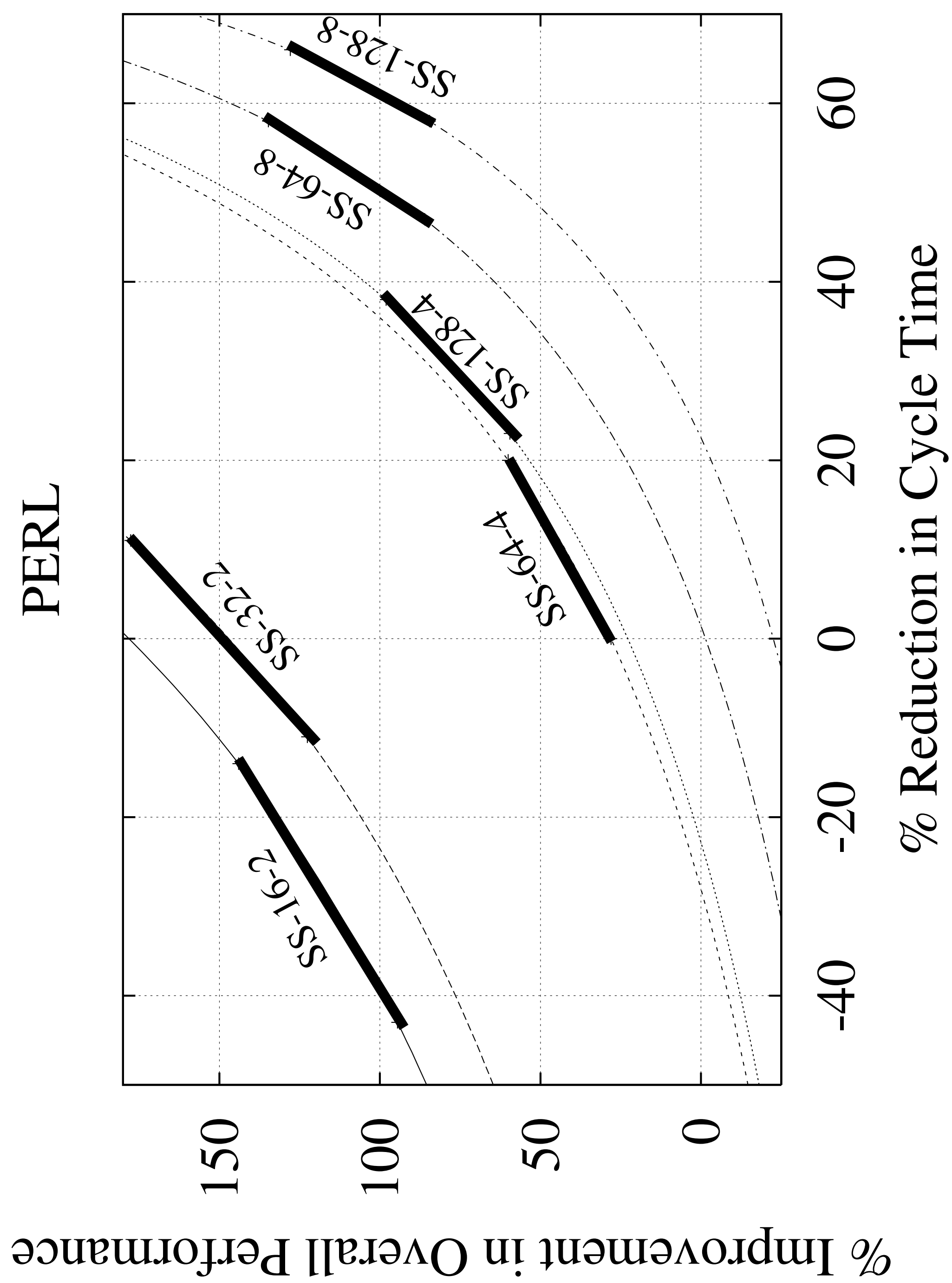


Figure 5-30: TP-128-16(16/8/2) vs. six superscalar processors (*perl*).

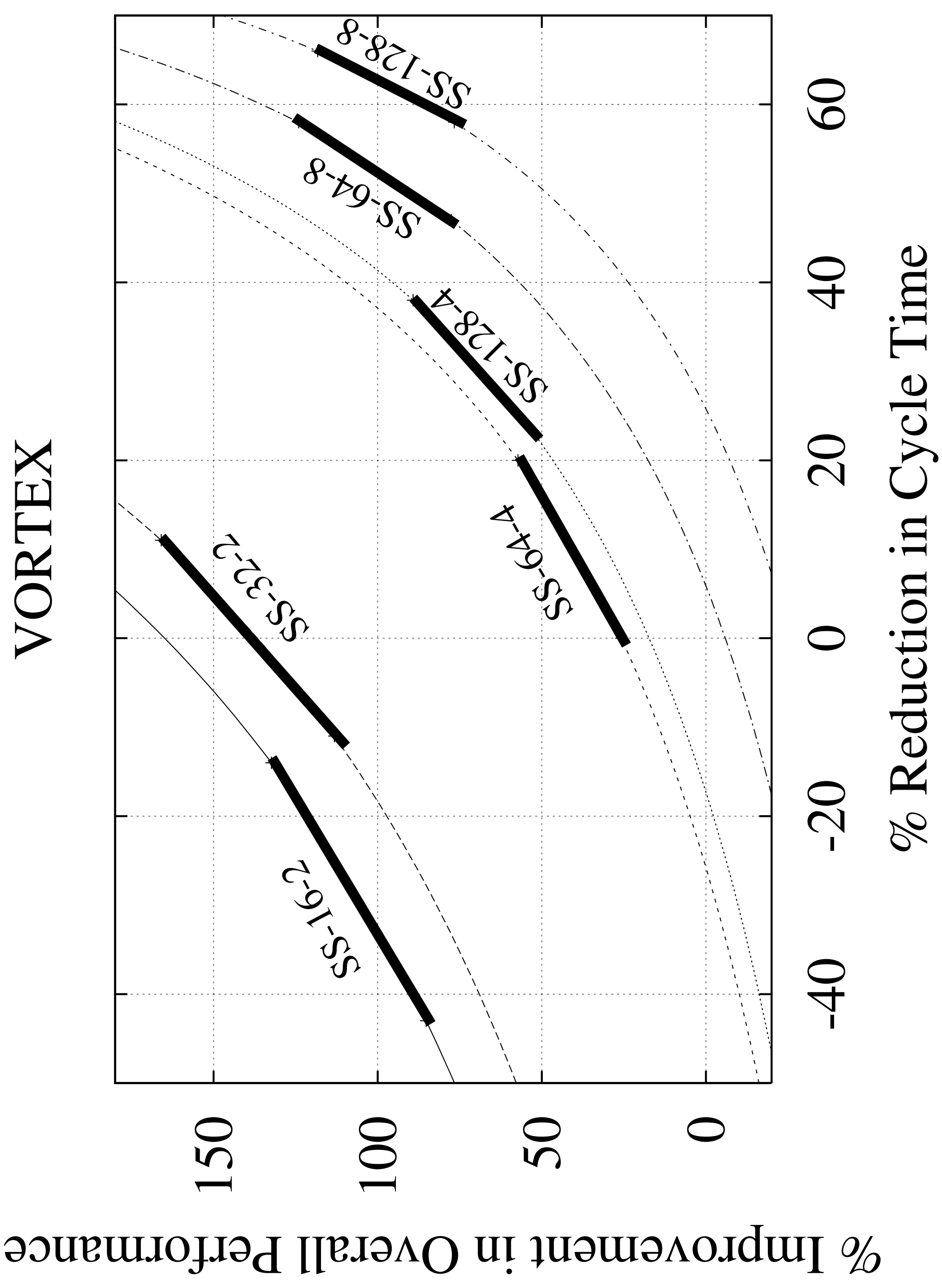


Figure 5-31: TP-128-16(16/8/2) vs. six superscalar processors (*vortex*).

5.2.4.4 Caveats

The previous study has the following shortcomings, and results should be interpreted with these shortcomings in mind.

1. Global bypass delay for the trace processor is only a rough estimate.

2. The analytical model for bypasses does not account for periodic buffering (repeaters) along long wires. The model, therefore, exaggerates quadratic effects.

3. Select delay calculations are potentially suspect because of the assumption that multiple functional units require serialized arbitration. After drawing out the logic for an arbiter node, it is clear that placing two arbiters in series is better than complicating the arbiter logic: the simple arbiter has one level of logic per node, and a complex arbiter necessarily has two levels of logic at each node. But it is unclear what happens to the complex arbiter as the number of functional units increases from two to four.

4. The study assumes cycle time is set by wakeup/select delays and/or bypass delays. But another part of the processor may dictate cycle time, such as the register file, caches, etc. Alternatively, even if cycle time is a function of wakeup/select/bypass delays, keep in mind that certain latencies remain constant -- for example, the cache access latency may be fixed and, therefore, may require more cycles as cycle time decreases. My simulations are contrary to this, because the number of cycles for a cache hit is fixed.

5.3 Three dimensions of trace processors

The three "dimensions" of trace processors are explored in this section: trace length (PE window size), number of PEs, and PE issue width. The study is organized into two parts. In the first part, only length-16 traces are considered and the relative importance of number of PEs and PE issue width is investigated. Trace length is deferred to the second part.

5.3.1 Number of PEs versus PE issue width

In this subsection, the relative importance of number of PEs and PE issue width is investigated for length-16 trace processors. My general approach is to determine when it is more beneficial to increase the number of PEs or PE issue width. Refer to the IPC graphs in Figures 5-32 through 5-38. The analysis pertains to the four curves labeled 16/*.

For a small number of PEs, doubling the number of PEs is more beneficial than doubling or even quadrupling the PE issue rate. This is particularly true for 2 PEs because the window is too small to effectively utilize a large aggregate issue bandwidth -- an overall window of 32 instructions simply does not expose enough ILP.

For 4 or more PEs, however, it is more beneficial to double the PE issue rate from 1-way issue to 2-way issue than to double the number of PEs. The trend is more noticeable as the overall window increases in size. Larger trace processors naturally expose more instruction-level parallelism and single-issue PEs are too weak to exploit this potential (recall the load balance observations in Section 5.2.2). But for 2-way issue PEs, it again becomes more beneficial to double the number of PEs than to double the issue rate. This is clearly visible in the graphs as a leveling off of the curves for length-16 traces. Evidently, a length-16 trace is too small a “window” to effectively utilize an issue bandwidth of 4.

In summary, if we are restricted to change only one dimension at a time, following is the sequence of configurations that maximize the incremental performance improvement at each step. (This trend applies to all of the benchmarks.)

$$16/2/1 \xrightarrow{\text{\# PEs}} 16/4/1 \xrightarrow{\text{issue/PE}} 16/4/2 \xrightarrow{\text{\# PEs}} 16/8/2 \xrightarrow{\text{\# PEs}} 16/16/2 \xrightarrow{\text{issue/PE}} 16/16/4$$

5.3.2 Trace length

Trace length is a complex dimension. It impacts the trace processor in so many ways, some of which were enumerated in Section 3.1.4. Trace length affects two essential aspects of the distributed instruction window.

1. *Load balance* - The size of the PE window affects the distribution of work and instruction-level parallelism among PEs.

2. *Global versus local complexity* - The number of global values decreases with longer traces (assuming constant window size). Firstly, this means fewer values incur the global bypass delay. Secondly, there is a general shift from global *processor-level complexity* to local *PE-level complexity*. Physically constructing an equivalent instruction window requires connecting fewer PEs together. For the same IPC performance, potentially fewer global result buses are needed, and the global register file may be smaller (both in terms of registers and write ports). Wakeup delay and bypass delay are reduced

due to fewer global tags and a smaller global file, respectively. Of course, the reduced global complexity is accompanied by an increase in local complexity: the PE window, local register file (size and ports), and PE issue width are all affected by the longer traces.

The graphs in Figures 5-32 through 5-38 show performance (IPC) for two trace lengths, 16 and 32 instructions. The global bypass latency is one cycle. All configurations use unconstrained global result buses and global register files to avoid a per-configuration search of the design space. Furthermore, complexity (or cycle time) is not considered in my somewhat limited analysis of trace length. Thus, only a few of the issues discussed above are covered by the analysis -- load balance and global bypass latency -- both of which are captured by IPC.

I begin by considering trace processors of equal dimensions except for trace length, i.e. the same number of PEs and issue width per PE. Refer to any of the graphs in Figures 5-32 through 5-38. For single-issue PEs, length-32 traces almost always perform worse than length-16 traces. Thus, in spite of the fact that total issue bandwidth is the same and the length-32 trace processor has twice the total window size, the length-16 trace processor wins. Single-issue PEs are sensitive to load balance, and length-32 traces exacerbate the problem by moving even more work into a single PE.

I hypothesize that parallelism is often unevenly distributed among instructions in the window, in which case it is better to divide the window finely (i.e. shorter traces). Coarsely dividing the instruction window localizes “clusters” of parallelism within individual PEs.

Also, branch mispredictions in either case limit the effective size of the window -- but given an identically-located misprediction, the coarser window provides less effective issue bandwidth to instructions before the mispredicted branch.

As issue width is increased, however, there is a crossover point and length-32 traces perform better. For *go*, *jpeg*, *m88ksim*, and *perl*, the crossover point is consistently at 2-way issue. At 4-way issue, the performance gap between length-32 and length-16 traces is even wider, and all benchmarks except *vortex* show better performance with length-32 traces. Increasing PE issue bandwidth reduces sensitivity to load balance, and the length-32 trace processor is able to exploit its larger instruction window and more localized communication to exceed length-16 trace processor performance.

Next, I compare length-16 and length-32 trace processors that have the same aggregate window size and issue bandwidth. Thus, for each length-16 trace processor configuration, the corresponding length-32 trace processor has half the number of PEs and twice the issue bandwidth per PE, as shown in Table 5-12. The comparison determines whether it is better to have longer or shorter traces for a fixed amount of hardware parallelism (considering only IPC and not cycle time).

Referring to Table 5-12, three processor pairs perform similarly for most of the benchmarks: `TP-64-4(16/4/1)` and `TP-64-4(32/2/2)`, `TP-64-8(16/4/2)` and `TP-64-8(32/2/4)`, and `TP-128-16(16/8/2)` and `TP-128-16(32/4/4)`. For the other three processor pairs, length-32 traces consistently perform better than length-16 traces.

Table 5-12: Comparing equivalent length-16 and length-32 trace processors.

Equivalent Trace Processor Configs.	gcc	go	jpeg	li	m88k	perl	vortex
TP-64-4(16/4/1) TP-64-4(32/2/2)	=	=	= (32+)	=	32 > 16	=	= (32+)
TP-64-8(16/4/2) TP-64-8(32/2/4)	=	= (16+)	= (32+)	= (16+)	32 > 16	= (16+)	=
TP-128-8(16/8/1) TP-128-8(32/4/2)	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16
TP-128-16(16/8/2) TP-128-16(32/4/4)	=	=	32 > 16	=	32 > 16	=	32 > 16
TP-256-16(16/16/1) TP-256-16(32/8/2)	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16
TP-256-32(16/16/2) TP-256-32(32/8/4)	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16	32 > 16

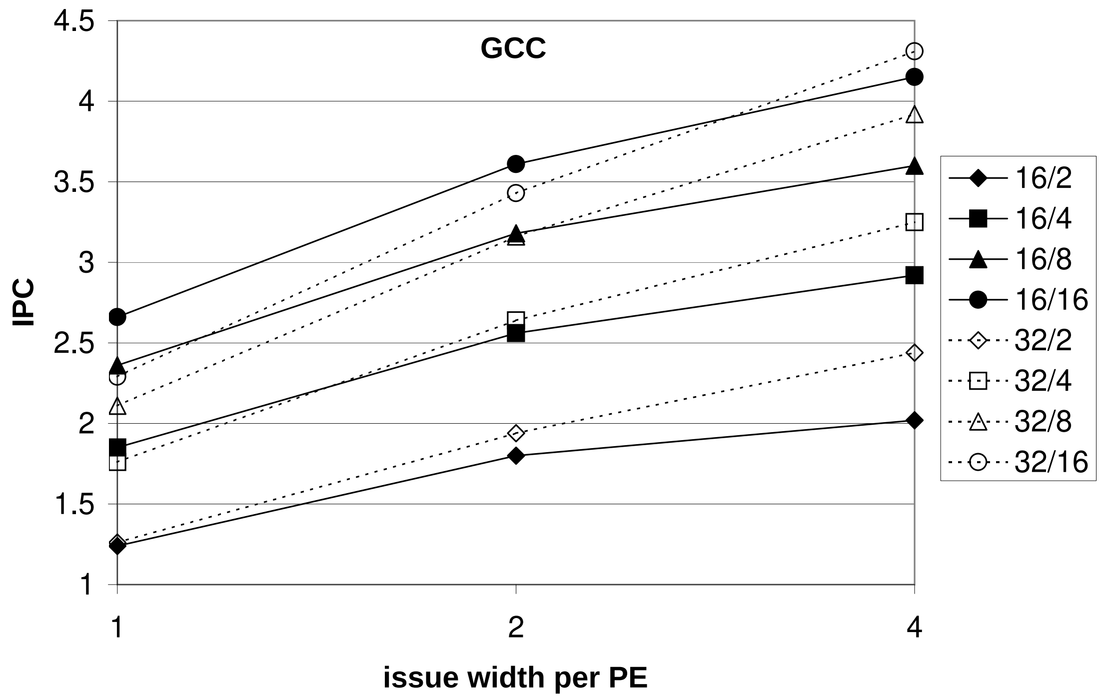


Figure 5-32: Varying the three trace processor dimensions (*gcc*).

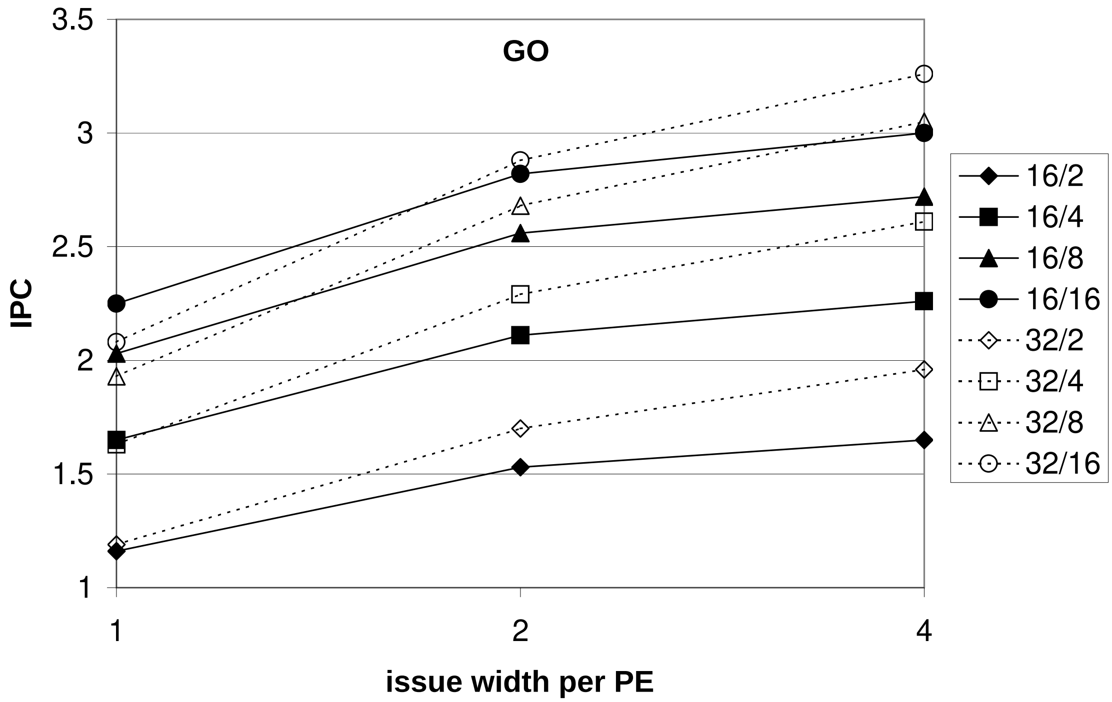


Figure 5-33: Varying the three trace processor dimensions (*go*).

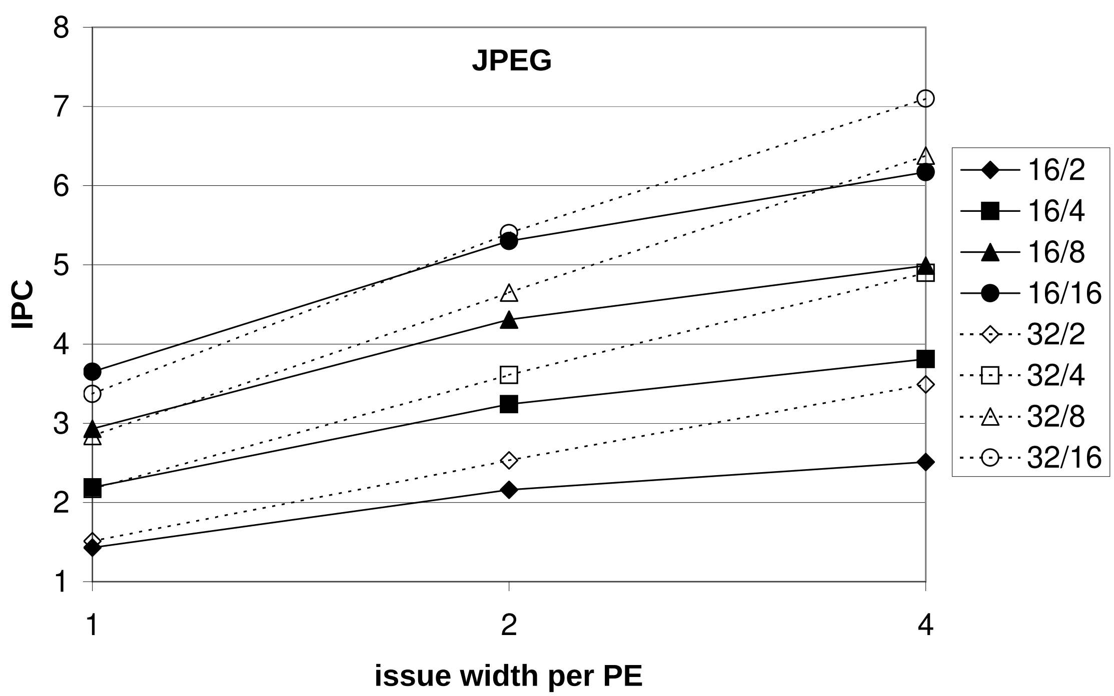


Figure 5-34: Varying the three trace processor dimensions (*jpeg*).

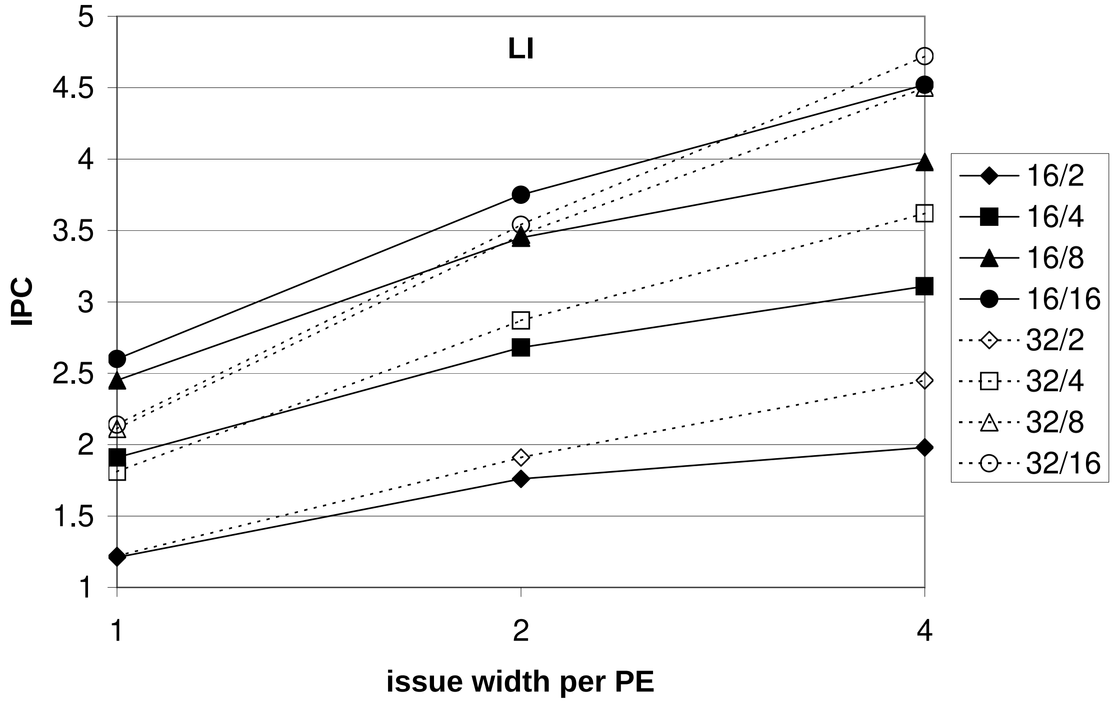


Figure 5-35: Varying the three trace processor dimensions (*li*).

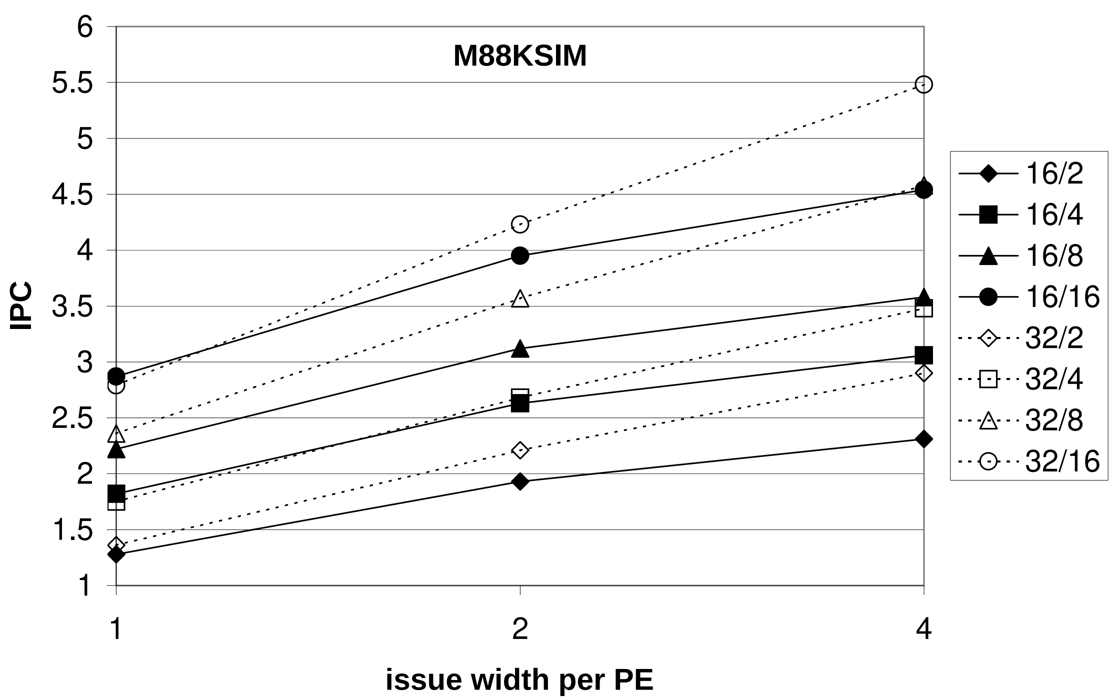


Figure 5-36: Varying the three trace processor dimensions (*m88ksim*).

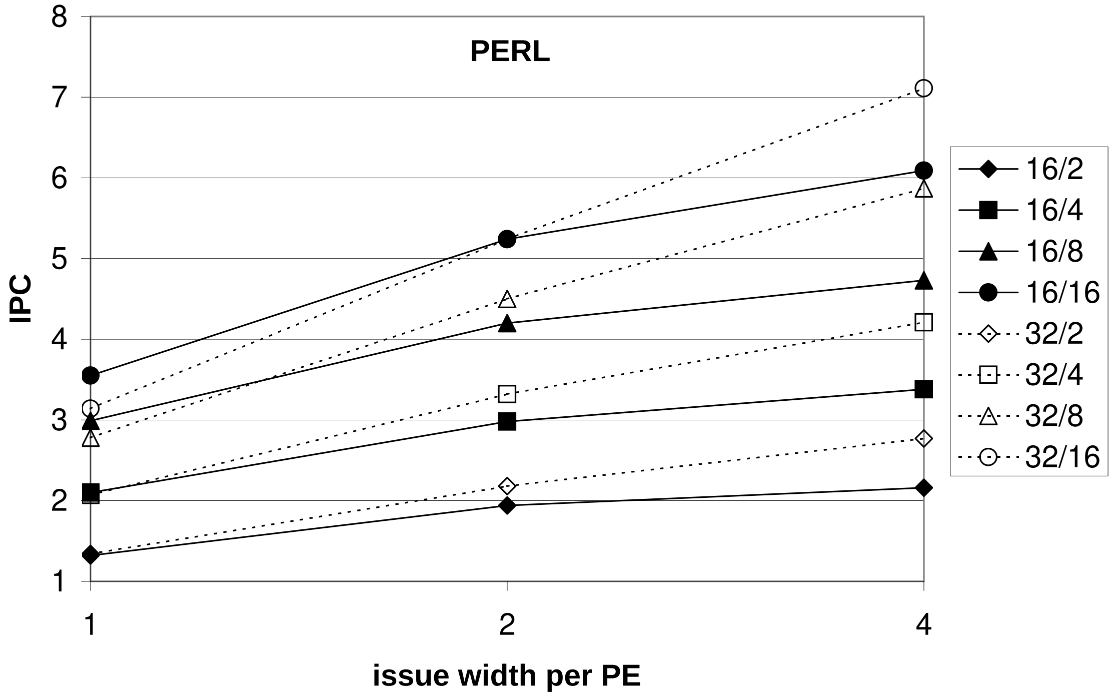


Figure 5-37: Varying the three trace processor dimensions (*perl*).

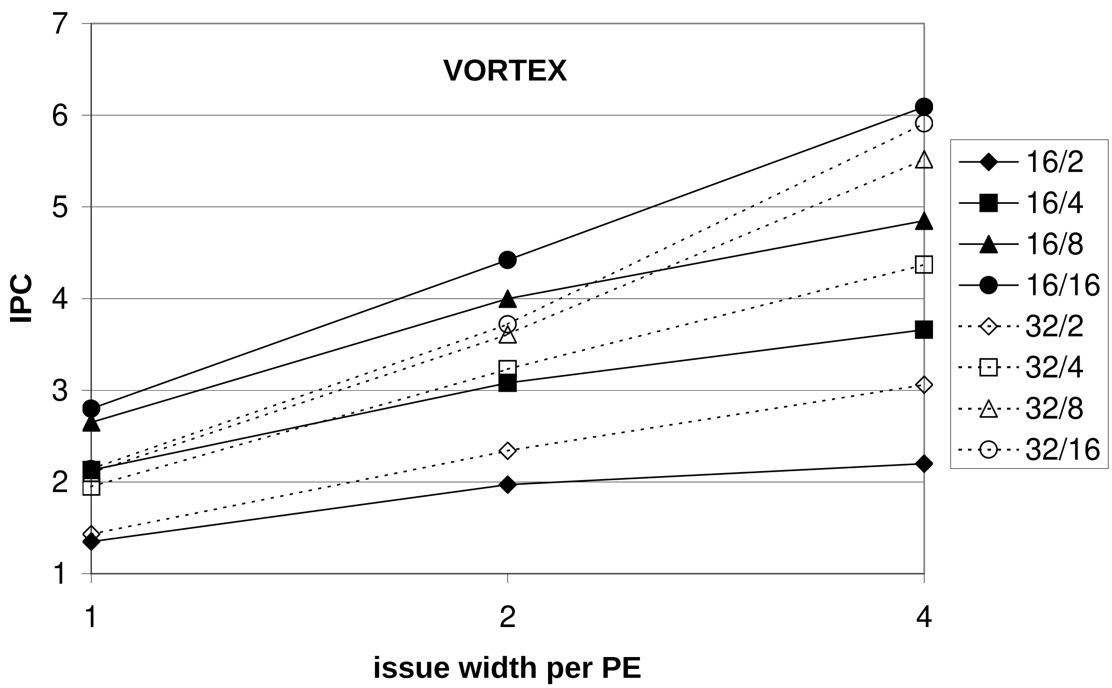


Figure 5-38: Varying the three trace processor dimensions (*vortex*).

5.4 Hierarchical register mechanism

The hierarchical division of registers is presumably beneficial because fewer values are globally communicated. The global register file size and write ports are reduced, fewer global result buses are required, and fewer values incur the longer global bypass latency. In this section, I 1) quantify the aforementioned benefits of the hierarchical register mechanism, 2) demonstrate that the benefits only increase with longer traces, and 3) determine global parameters that perform as well as unconstrained global resources.

5.4.1 Register communication bandwidth

The graphs in Figures 5-39 through 5-45 plot performance as a function of global result bus bandwidth. I model 2, 4, and 6 global result buses. Also, to measure the benefit of register hierarchy, there are two register models. The first treats all values as globals, i.e. both local and global values must be communicated via the global result buses. This model is labeled "non-hier" (non-hierarchical) in the graphs. The second model uses hierarchical communication and is labeled "hier" in the graphs. In all experiments, the latency to communicate local and global values is uniform and fast, in order to isolate only bandwidth aspects.

Four trace processor configurations are simulated. The configurations were chosen to represent two window sizes, 128 and 256, since I expect bandwidth requirements to increase with the overall level of hardware parallelism. For each window size, equivalent length-16 and length-32 trace processors are represented. The `16/8/2` and `32/4/4` trace processors tend to perform similarly, as do `16/16/2` and `32/8/4` to a lesser extent, so I can effectively quantify the impact of trace length on global result bus bandwidth requirements.

Several trends are evident from the graphs in Figures 5-39 through 5-45. For a given number of result buses, register hierarchy significantly improves performance. With 2 result buses and no hierarchy, IPC saturates at a low value for all benchmarks and all configurations. As one might expect, the saturation point is between 2.0 and 3.0 IPC since throughput is dictated by the result bus bottleneck (greater than 2 IPC is possible because not all instructions write a destination register). The hierarchical model offloads much of the register write traffic to local result buses, and interestingly, having only 2 global result buses for a 128-instruction or 256-instruction window is a reasonable cost-performance design point.

One can interpret this trend another way: for the same level of performance, the hierarchical model requires fewer global result buses than the non-hierarchical model. Without hierarchy, doubling the number of result buses from 2 to 4 eliminates a significant bottleneck and performance sufficiently increases. But a similar effect can be achieved with fewer result buses if hierarchy is exploited. For example, *go* has nearly identical performance with "4 buses (non-hier)" and "2 buses (hier)".

Performance with unconstrained result bus bandwidth is not explicitly shown in the graphs of Figures 5-39 through 5-45, however, the IPCs for "6 buses (hier)" are essentially the same as the unconstrained IPCs reported in Section 5.2.2. Without hierarchy, increasing the number of result buses from 4 to 6 achieves nearly the same performance as unconstrained bandwidth. The only notable exception is *jpeg*, for which even "6 buses (non-hier)" is not enough to achieve the performance of unconstrained bandwidth. With hierarchy, only 4 global result buses are needed to approach the performance of unconstrained bandwidth. Minor differences occur only in the 256 window configurations for *jpeg*, *perl*, and *vortex*. These minor differences are diminished with "6 buses (hier)".

Next, I discuss *window size* and *trace length* and their relation to global result bus bandwidth. The benchmark *gcc* is used as a representative example. Its sensitivity to result bus bandwidth is shown in Figure 5-46. The bottom and top parts of each bar show performance improvement as bandwidth is doubled from 2 to 4 and from 4 to 6, respectively.

- *Window size* - Larger windows are more sensitive to global result bus bandwidth. In Figure 5-46, the performance difference between "2 buses (hier)" and "4 buses (hier)" increases from 11% to 22% as the number of PEs is doubled from `16/8/2` to `16/16/2`.
- *Trace length* - Figure 5-46 clearly shows length-32 traces reduce sensitivity to global result bus bandwidth. The total bar height for configuration `32/4/4` is half that of the equivalent length-16 trace processor `16/8/2`. A trace length of 32 increases the ratio of local registers to global registers and, consequently, reduces global communication. Likewise, `32/8/4` is less sensitive to global result bus bandwidth than the equivalent length-16 trace processor `16/16/2`. However, `32/8/4` is more sensitive than `32/4/4` because it has a larger instruction window.

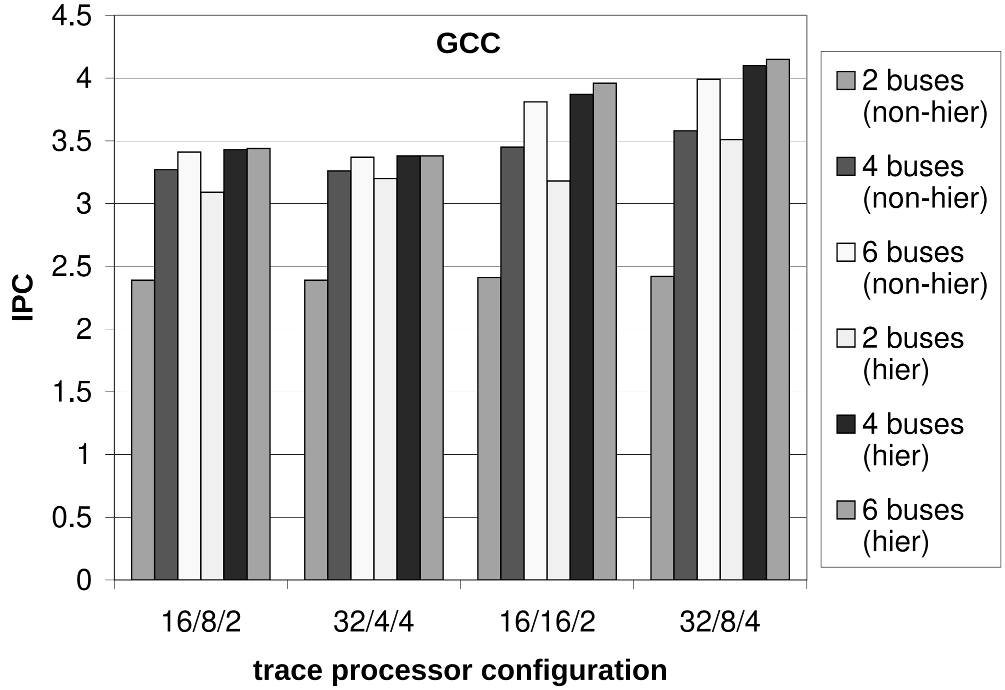


Figure 5-39: Global result bus experiments (*gcc*).

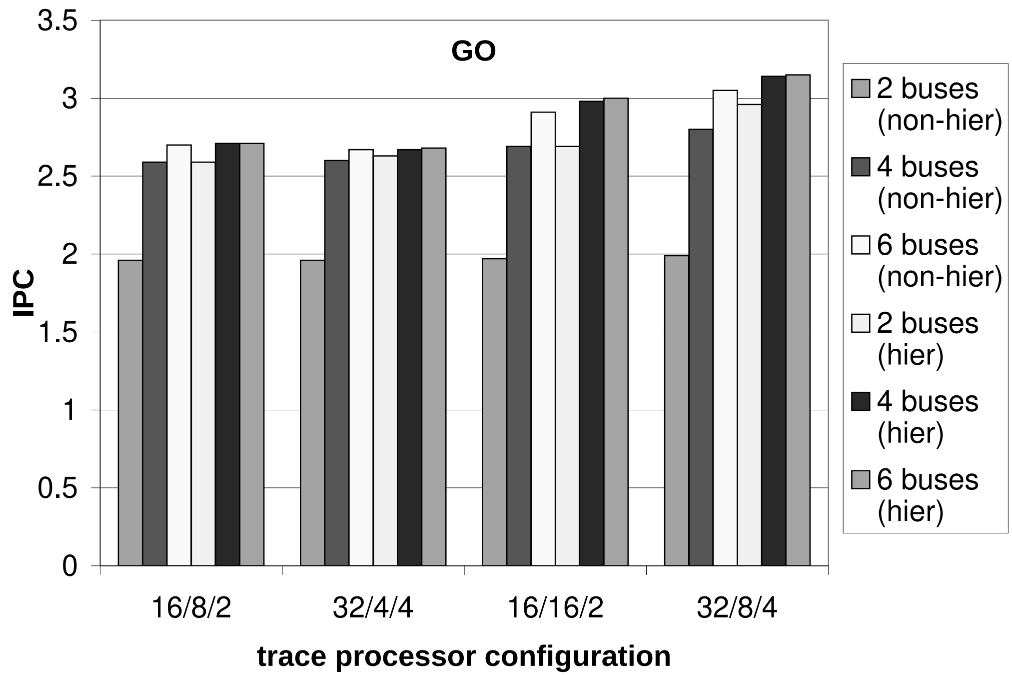


Figure 5-40: Global result bus experiments (*go*).

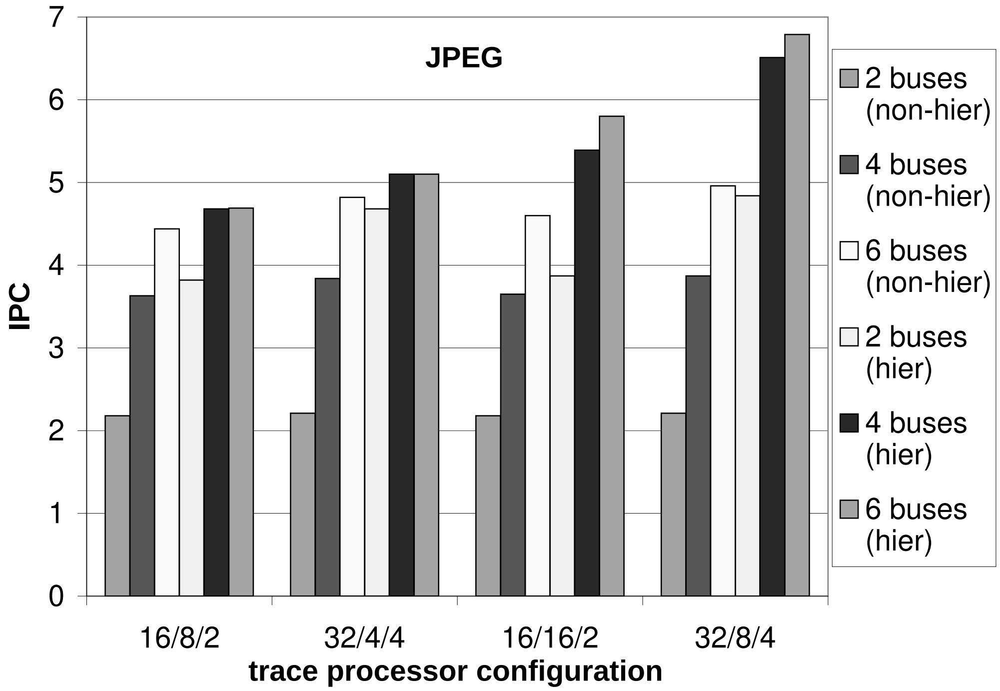


Figure 5-41: Global result bus experiments (*jpeg*).

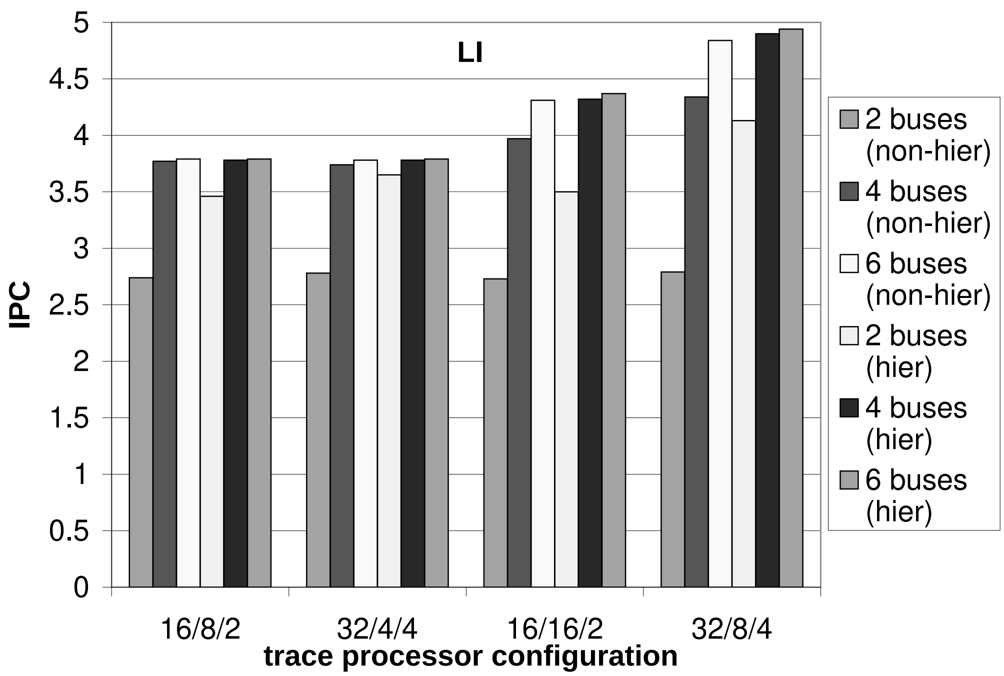


Figure 5-42: Global result bus experiments (*li*).

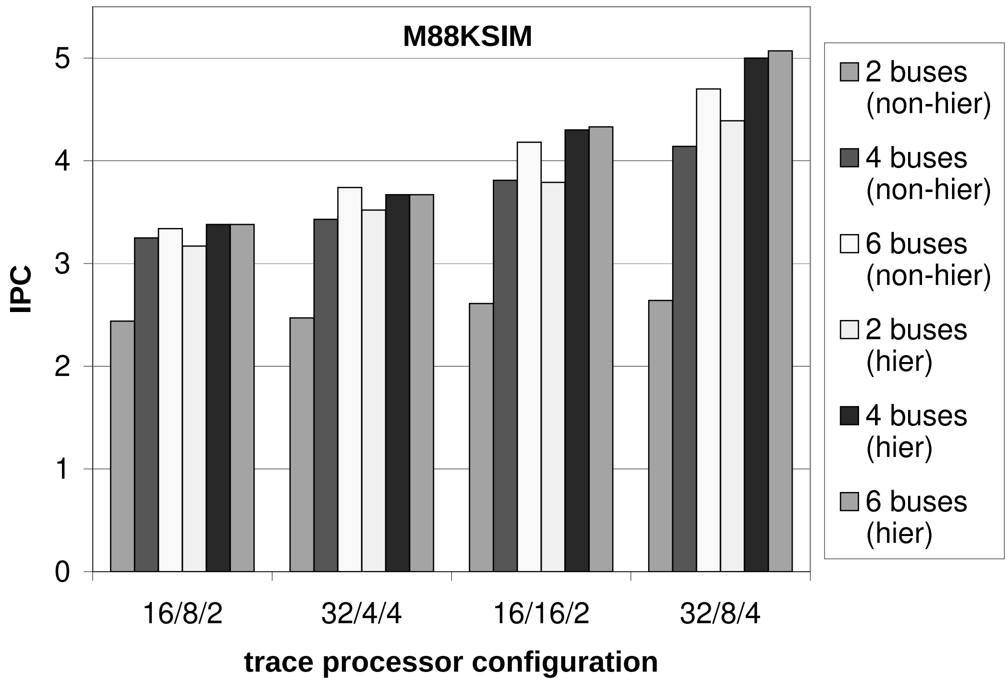


Figure 5-43: Global result bus experiments (*m88ksim*).

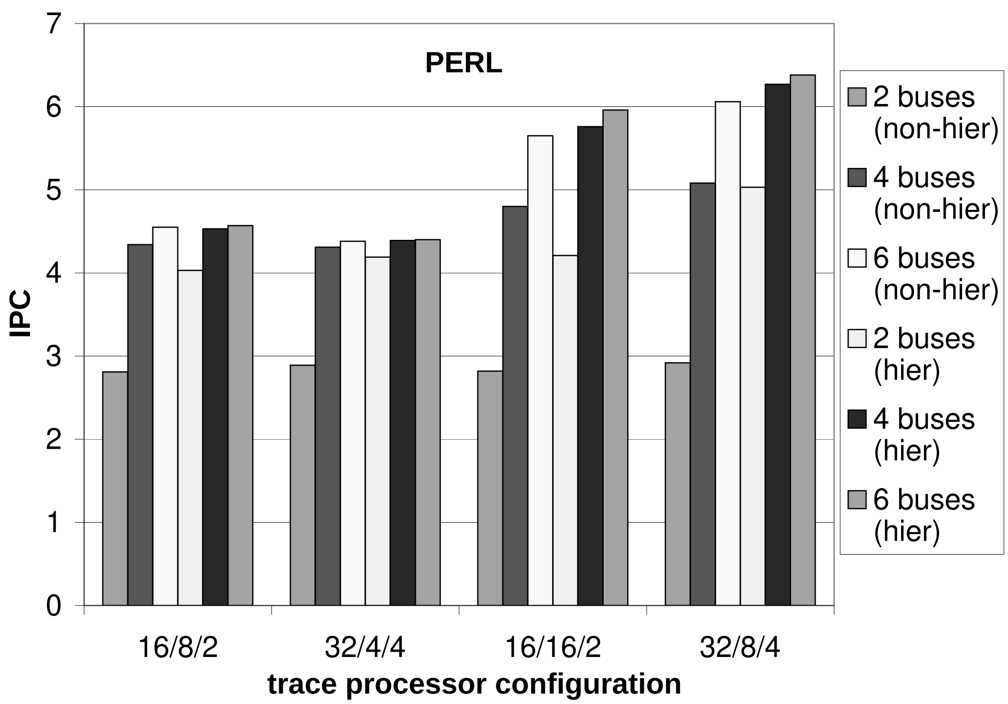


Figure 5-44: Global result bus experiments (*perl*).

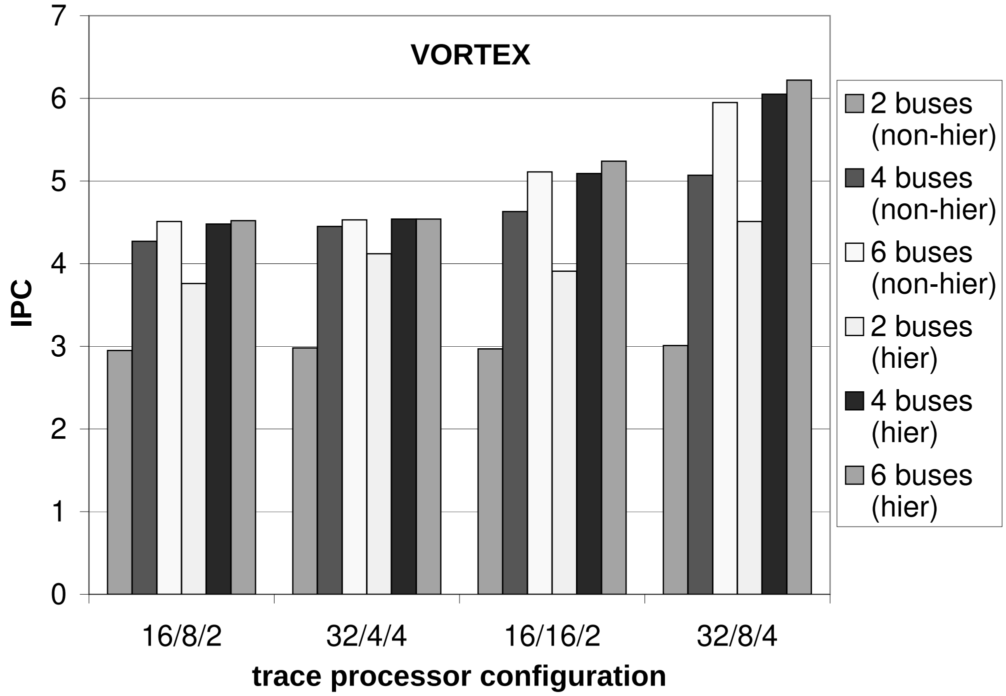


Figure 5-45: Global result bus experiments (*vortex*).

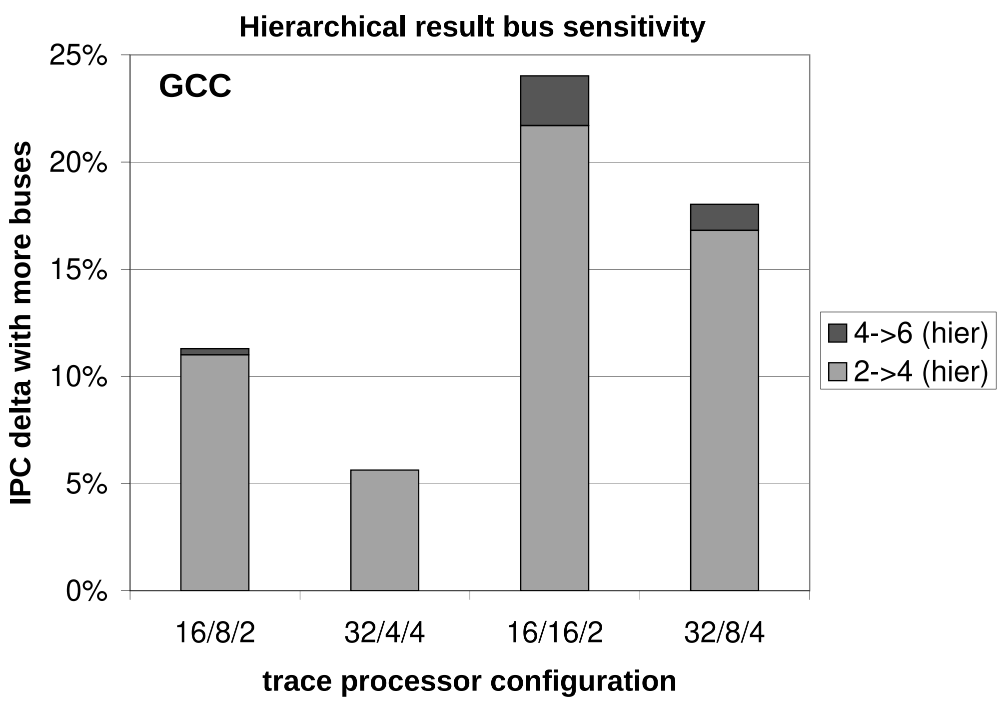


Figure 5-46: Global result bus sensitivity (*gcc*).

5.4.2 Register communication latency

I observed in Section 5.2.4 that operand bypass delay increases significantly with issue width and may become a cycle time limiter. There are two possible ways of dealing with bypass delay. Cycle time can be extended to accommodate the delay or, instead, we may opt for a short clock cycle and one or more cycles to perform operand bypassing.

Trace processors use the latter approach. However, hierarchy is exploited to reduce the number of values that actually incur the one or more cycles of communication delay. This section first evaluates the efficacy of hierarchy in reducing performance penalties due to operand bypass latency. I arrive at a "rule-of-thumb" number for estimating the IPC decrease caused by global bypass latency. Finally, I measure the sensitivity of four trace processor configurations to global bypass latency; latency is varied from one to three cycles to essentially model progressively longer chip crossings and shorter clock cycles. The four trace processor configurations are the same used in the previous section (i.e. two window sizes and two trace lengths); also, global result bus bandwidth is unconstrained in order to isolate the effect of latency.

The first graph in each of Figures 5-47 through 5-53 shows performance as a function of operand bypass latency. There are two latencies, local bypass latency and global bypass latency. Each bar in the graph is labeled with the pair <*local latency*>/<*global latency*>. A latency of 0 means result bypass is performed in the same cycle that the result is computed, i.e. data dependent instructions execute in back-to-back cycles as in Figure 3-7.

To quantify the efficacy of hierarchy, consider the two bars labeled "1/1" and "0/1". The bar "1/1" corresponds to a non-hierarchical model, i.e. values are uniformly penalized one full cycle for bypass. The “0/1” bar is a hierarchical model, i.e. only global values are penalized one full cycle for bypass. The performance decrease with respect to “0/0” for each of “1/1” and “0/1” is plotted in the second graph of Figures 5-47 through 5-53. Penalizing all values (“1/1”) results in a 20% to 35% decrease in IPC, depending on the benchmark. Hierarchy (“0/1”) is quite effective in reducing this performance loss by more than half (the few exceptions are the `16/16/2` data for *li* and *vortex*, but there is still a noticeable improvement). As a rule-of-thumb, a global bypass latency of 1 cycle decreases performance by about 10% for length-16 trace processors and 5% for length-32 trace processors.

The third graph in each of Figures 5-47 through 5-53 measures sensitivity to global bypass latency. Local bypass latency is 0 and global bypass latency is varied from 0 to 3 cycles. The graph plots decrease in IPC relative to 0 global bypass latency. Three general trends are evident.

1. Performance decreases nearly linearly with global bypass latency. This trend has serious implications -- there is no reprieve, i.e. no tapering-off effect with longer latency.

2. Length-32 trace processors are noticeably less sensitive to global bypass latency than equivalent length-16 trace processors, because length-32 traces reduce global communication more than length-16 traces (for an equal size window).

3. Somewhat surprisingly, larger instruction windows (256) are more sensitive to global bypass latency than smaller windows (128). One would have expected greater “latency-tolerance” with more hardware parallelism.

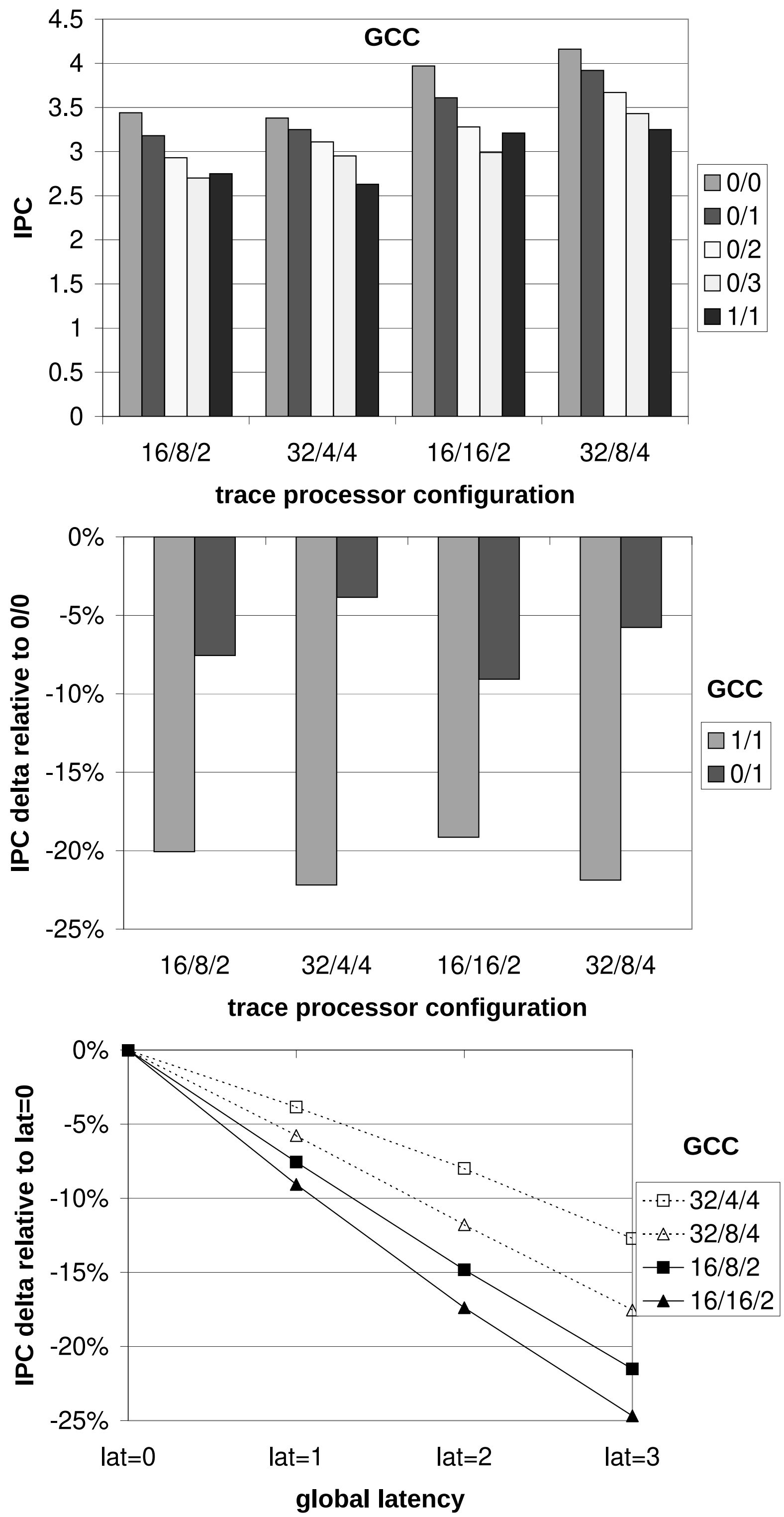


Figure 5-47: Global bypass latency experiments (*gcc*).

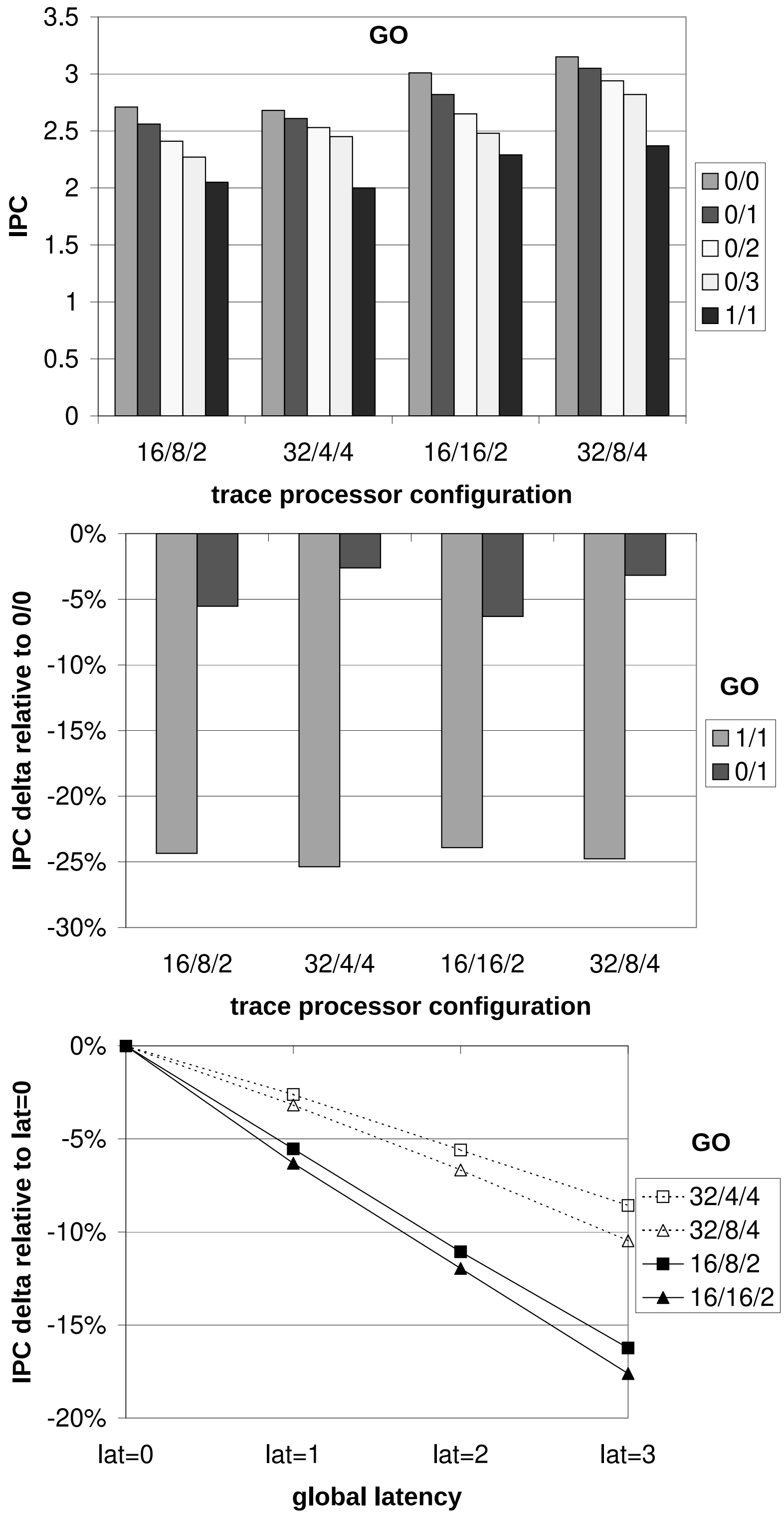


Figure 5-48: Global bypass latency experiments (*go*).

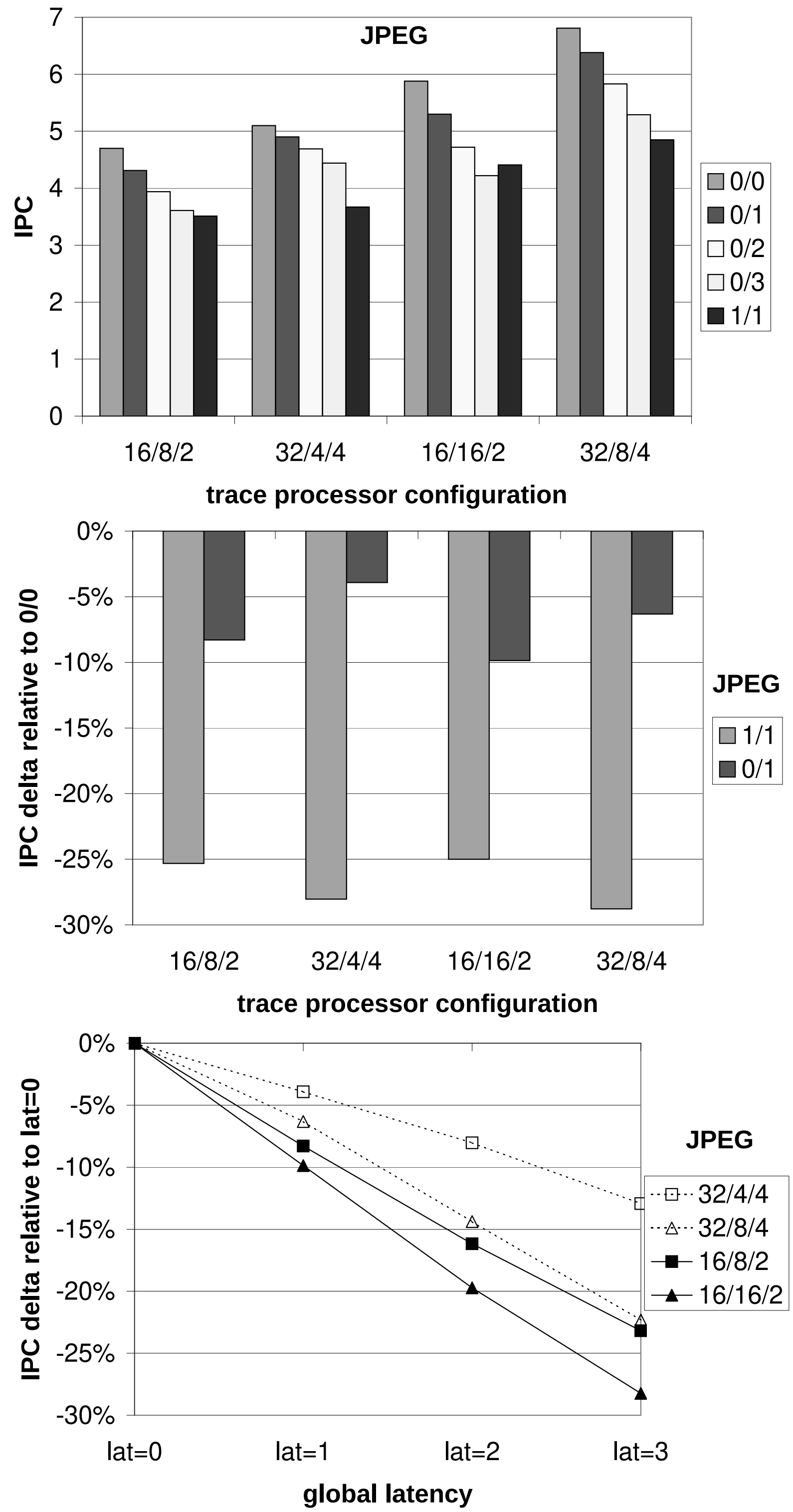


Figure 5-49: Global bypass latency experiments (*jpeg*).

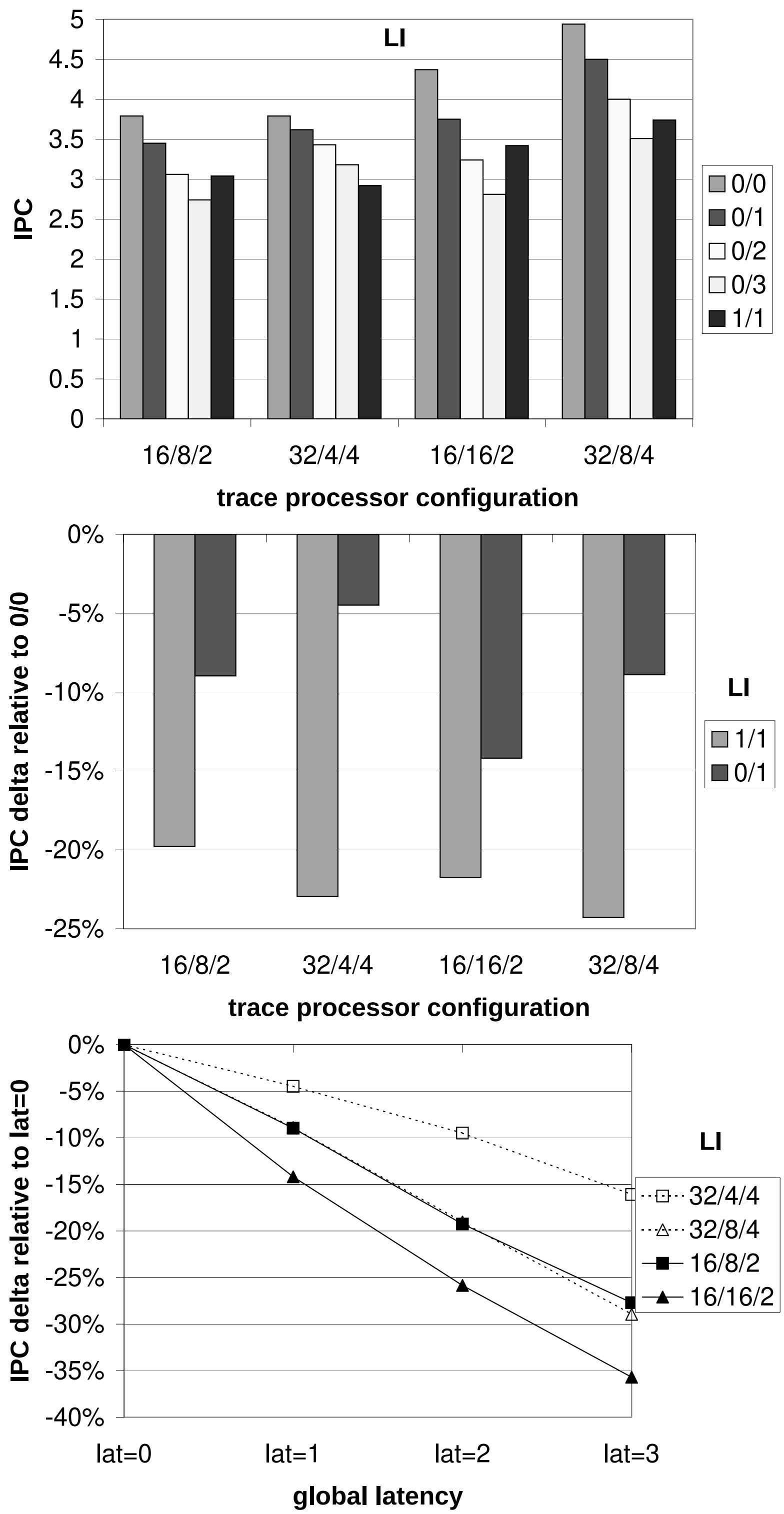


Figure 5-50: Global bypass latency experiments (*li*).

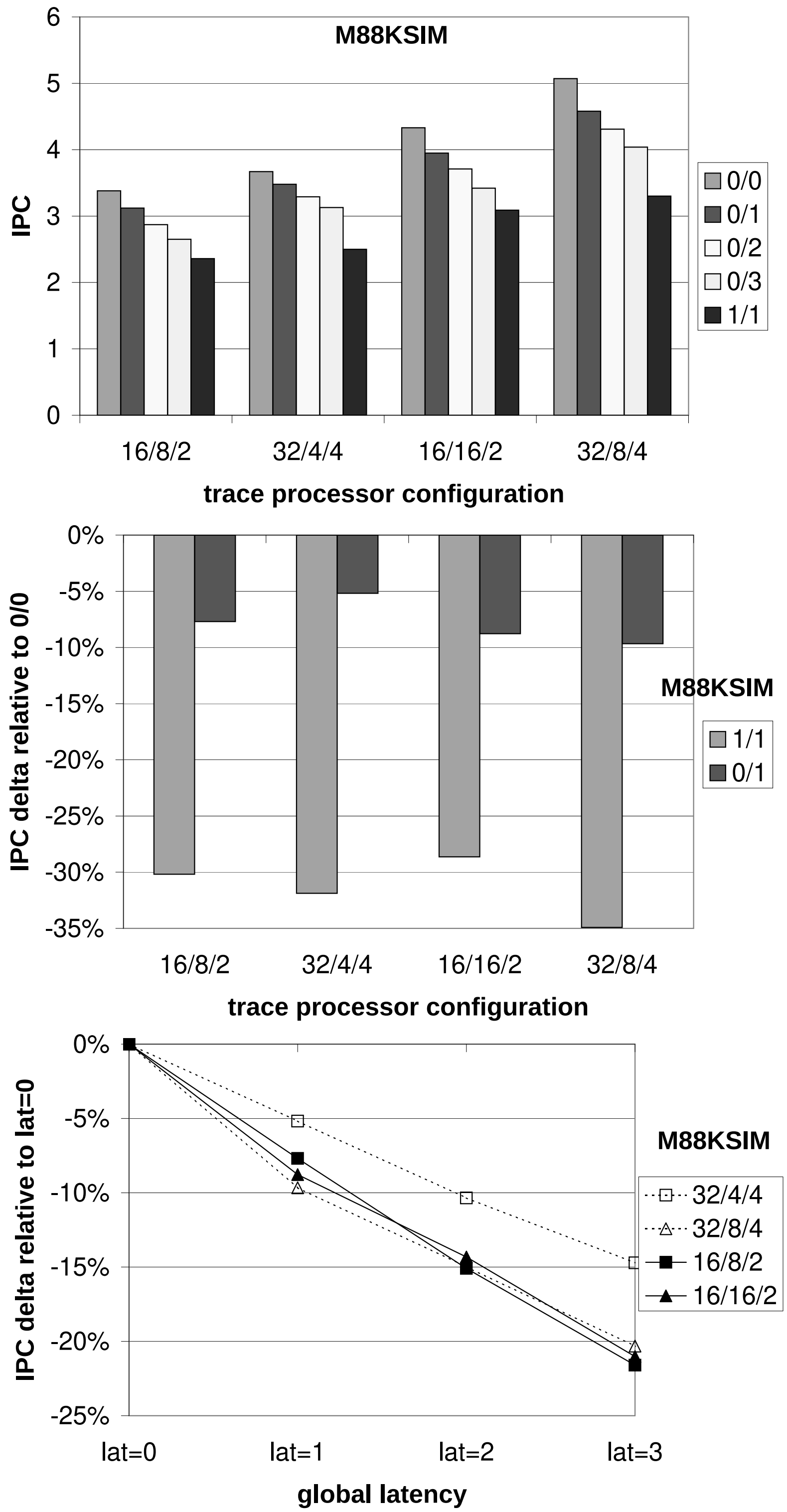


Figure 5-51: Global bypass latency experiments (*m88ksim*).

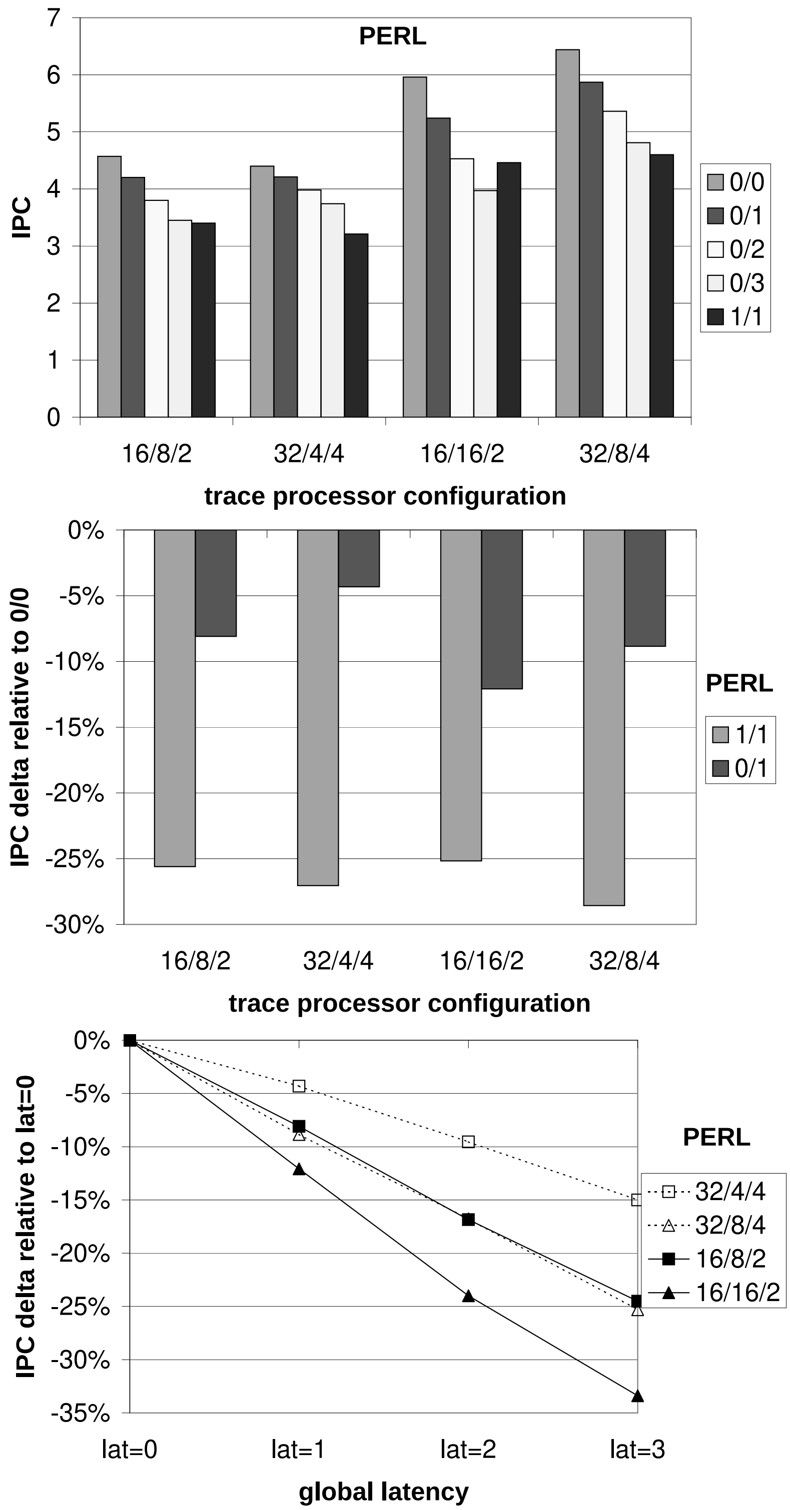


Figure 5-52: Global bypass latency experiments (*perl*).

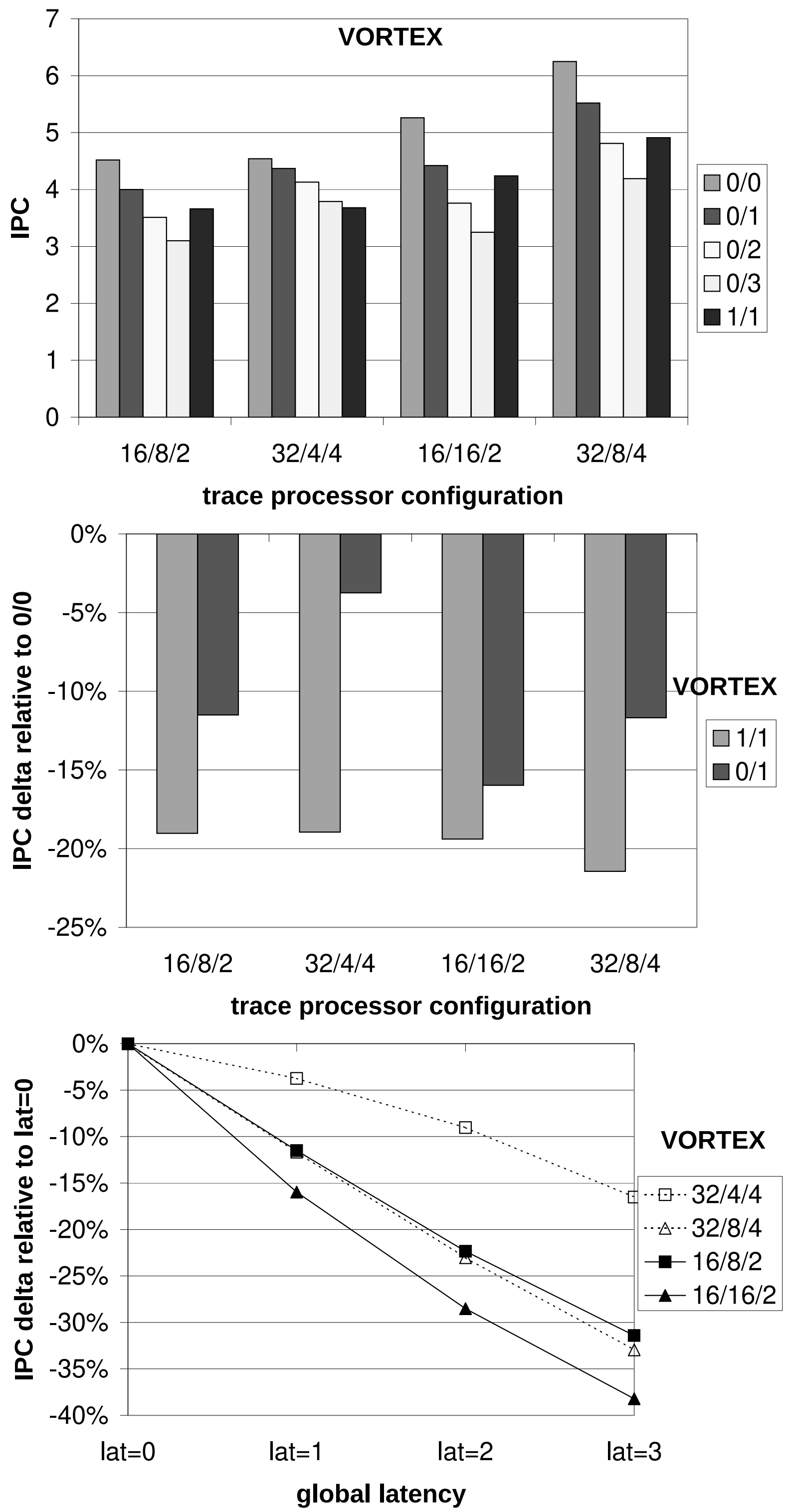


Figure 5-53: Global bypass latency experiments (*vortex*).

5.4.3 Global register file size

The trace processor's centralized global register file is smaller than the register file in an otherwise equivalent superscalar processor. The global register file stores only global values and local values are managed by distributed local register files.

The graphs in Figure 5-54 quantify the reduction in register file size. Two equivalent trace processor configurations are considered, one with a trace length of 16 (`TP-128-16(16/8/2)`) and the other with a trace length of 32 (`TP-128-16(32/4/4)`). For each configuration, there are two graphs; the first corresponds to a trace processor that exploits register hierarchy ("hier") and the second corresponds to a trace processor that does not exploit hierarchy ("non-hier"), i.e. all values are managed by the global register file. IPC is plotted as a function of the number of global registers. I start with 64 global registers and add registers in increments of 16. Performance with an unlimited number of global registers is also indicated.

The `16/8/2` configuration with hierarchy requires only 80 global registers to achieve the same performance as unlimited registers. Without hierarchy, however, the same processor requires at least 96 physical registers and possibly 112 or more.

The number of global registers is reduced even further with length-32 traces (assuming the same window size). The `32/4/4` configuration requires only 64 global registers to achieve the same performance as unlimited registers, although potentially fewer than that are required; I did not experiment with fewer than 64 registers.

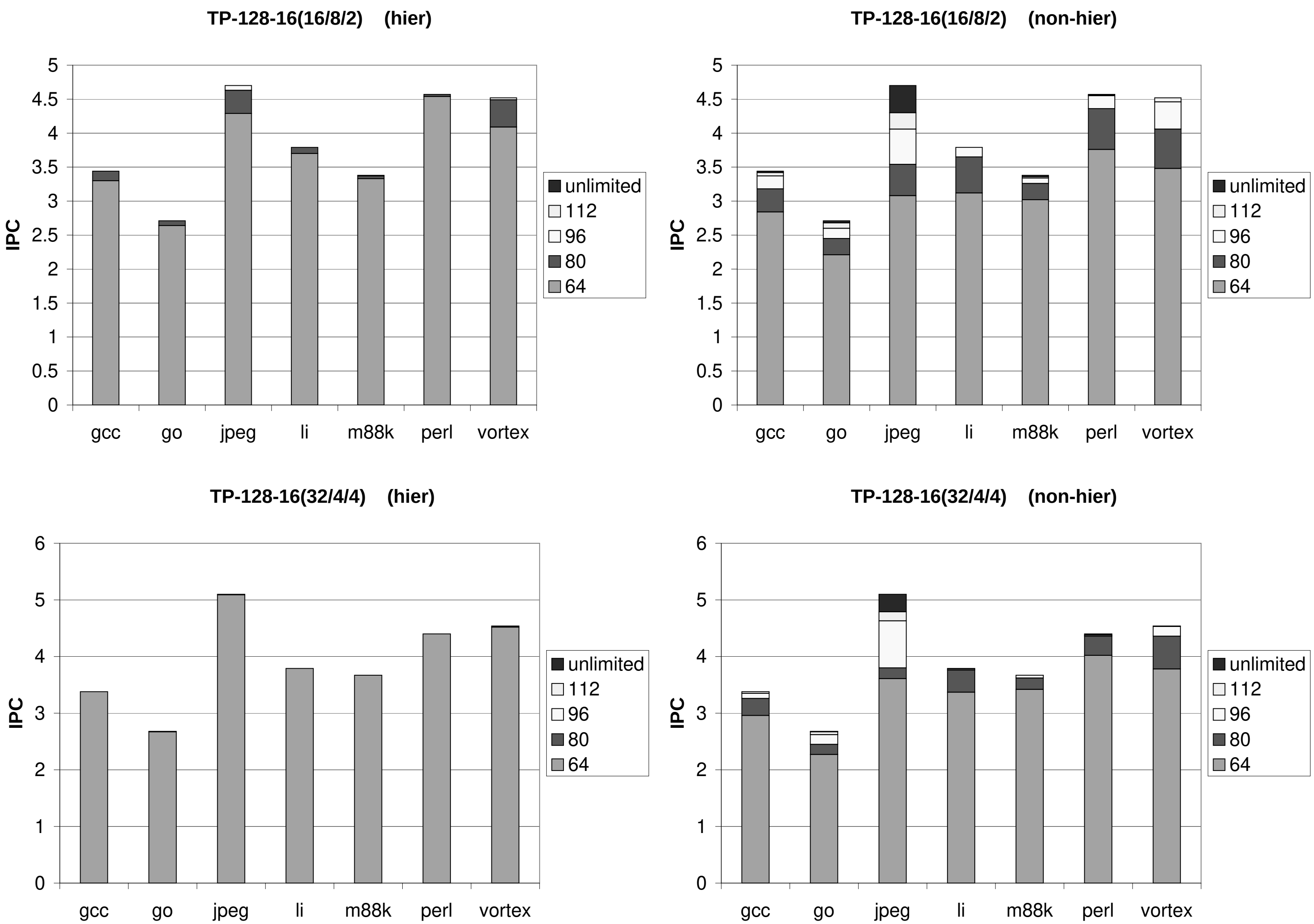


Figure 5-54: Implications of hierarchy and trace length on global register file size.

5.5 Floating-point performance

As mentioned in this chapter's introduction, full results for floating-point benchmarks were omitted primarily because their relatively regular control flow and parallelism make them amenable to a range of well-established ILP techniques, such as vector processing and software pipelining. Nevertheless, the primary performance results, as presented earlier in Section 5.2.3 for integer benchmarks, are given for floating point benchmarks in Figures 5-55 through 5-62. The graphs show IPC for both superscalar and trace processors (the trace processors require 1 cycle to bypass global operands).

Several related trends can be observed in floating-point benchmarks that distinguish them from integer benchmarks. Except for *tomcatv* and *wave5*, the trace processor curves are flatter than observed for integer benchmarks and, moreover, trace processors approach the performance of superscalar processors with the same window size. Flatter curves indicate less sensitivity to PE issue width, yet IPC increases substantially with more PEs, in almost the same manner that increasing superscalar window size increases IPC. For both these reasons, I conclude that parallelism is more evenly distributed among PEs for the floating-point benchmarks, so distributing the superscalar instruction window and partitioning issue bandwidth has less performance impact than shown previously. There is sufficient and evenly distributed parallelism such that *total window size* and *aggregate hardware throughput* (e.g. data cache bandwidth) are the primary performance limiters in floating-point benchmarks, not the distinction between distributed and centralized resources.

Another result, not explicitly shown in this section for brevity, is that floating-point benchmarks are less sensitive to global bypass latency than integer benchmarks. This was also observed in multiscalar processors [].

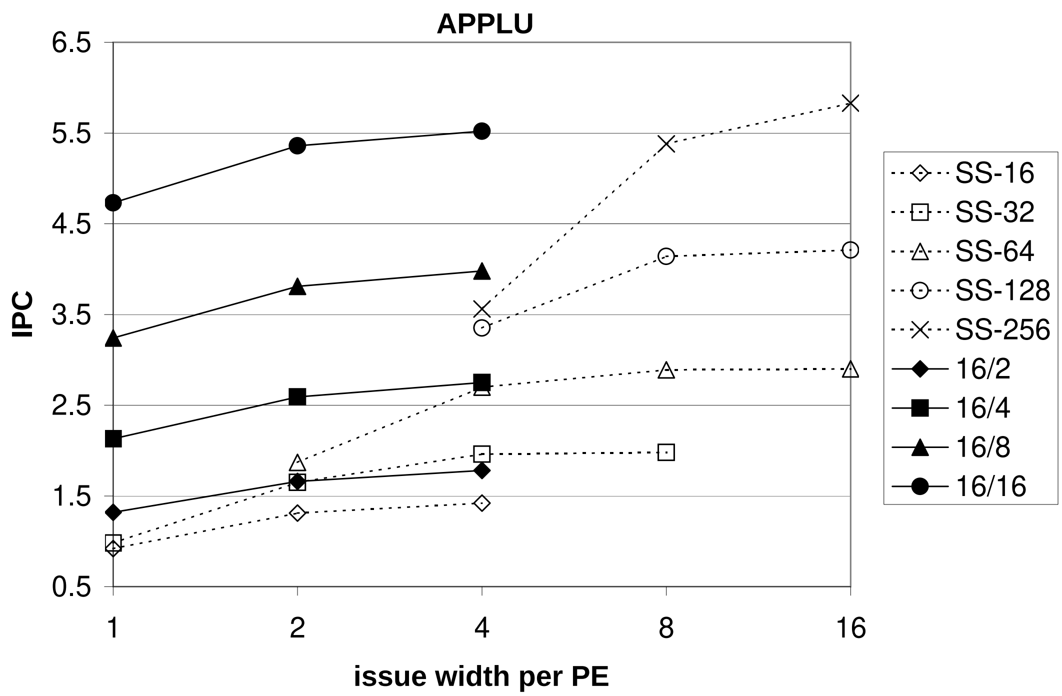


Figure 5-55: Superscalar vs. trace processors with partial bypasses (*applu*).

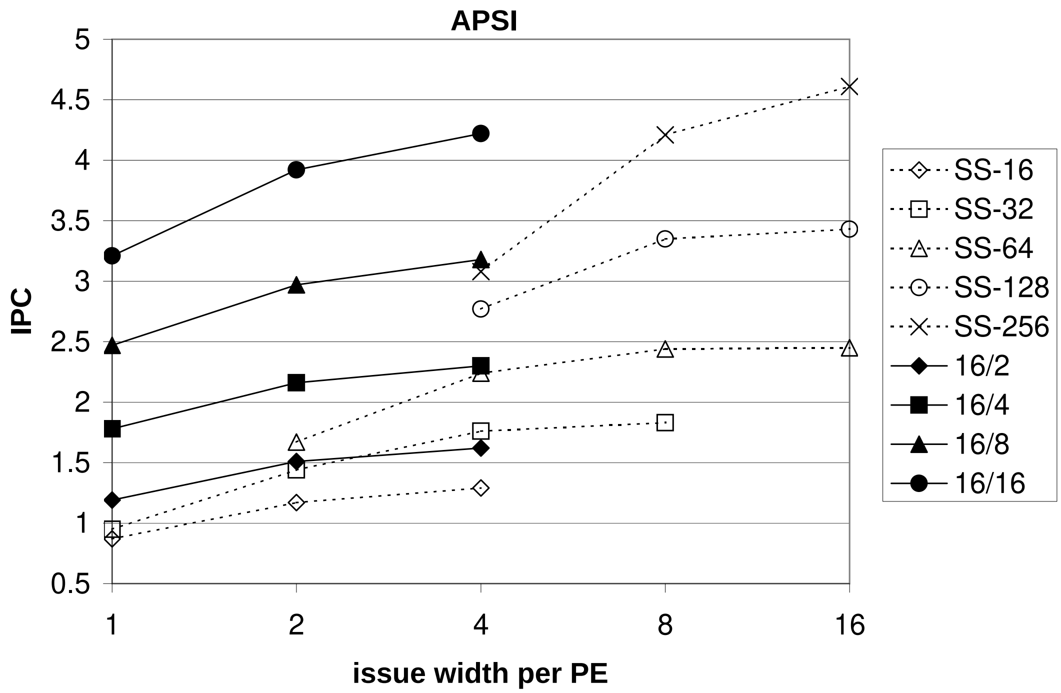


Figure 5-56: Superscalar vs. trace processors with partial bypasses (*apsi*).

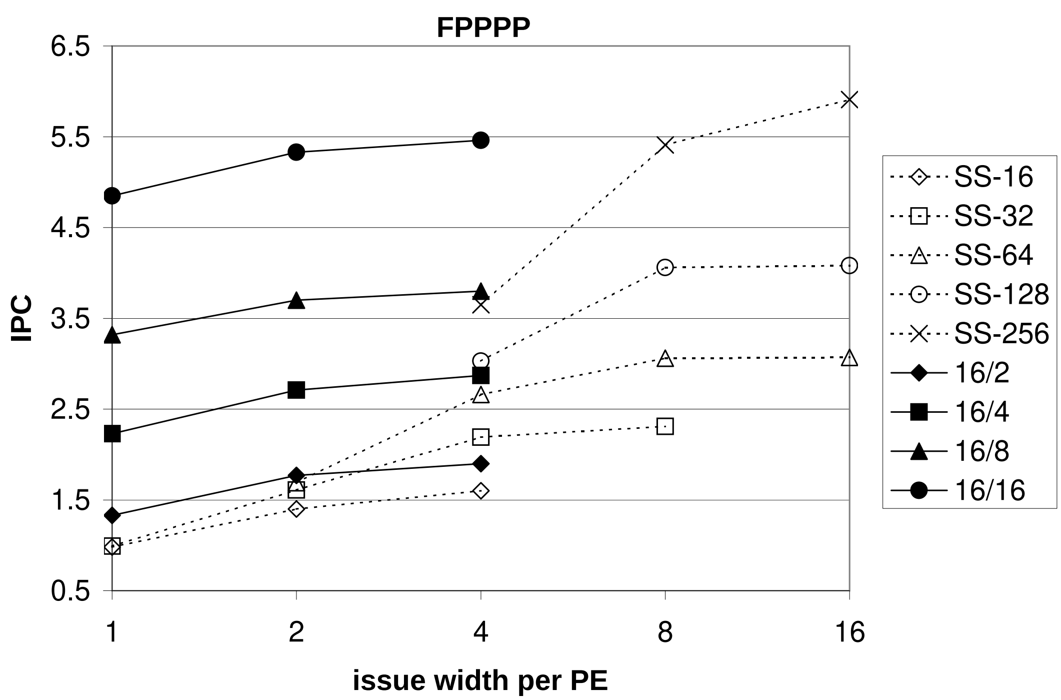


Figure 5-57: Superscalar vs. trace processors with partial bypasses (*fpppp*).

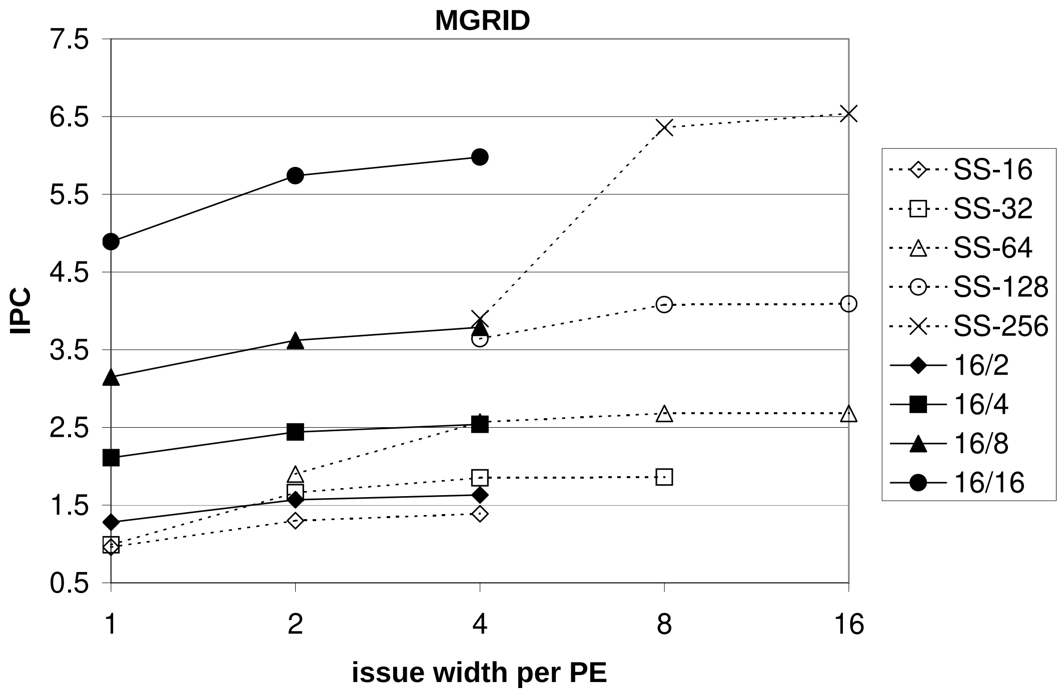


Figure 5-58: Superscalar vs. trace processors with partial bypasses (*mgrid*).

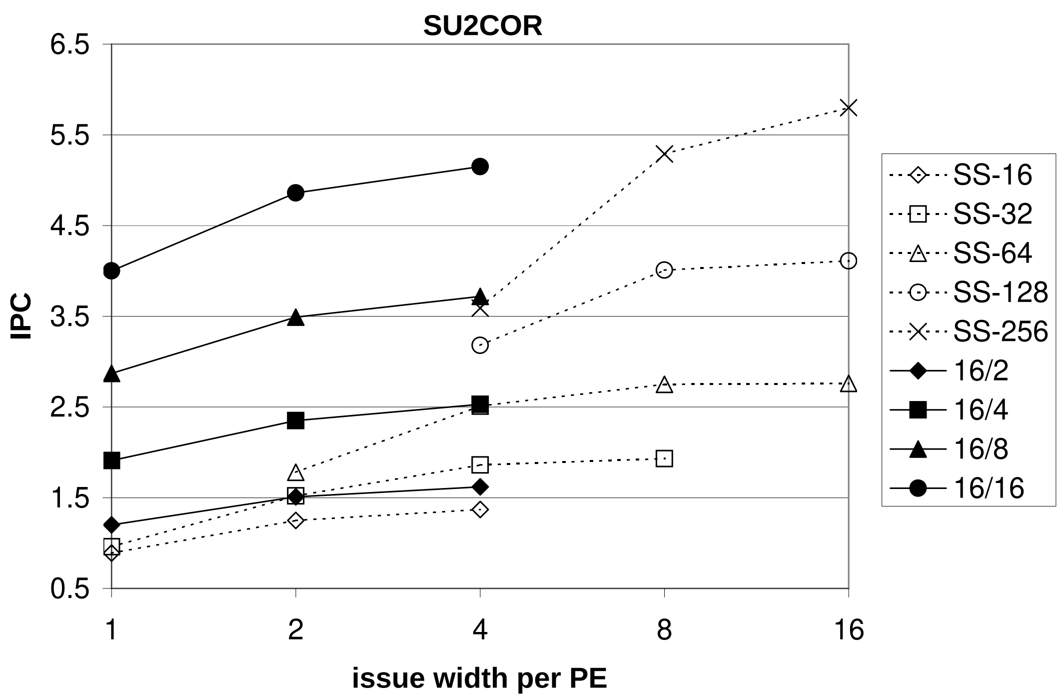


Figure 5-59: Superscalar vs. trace processors with partial bypasses (*su2cor*).

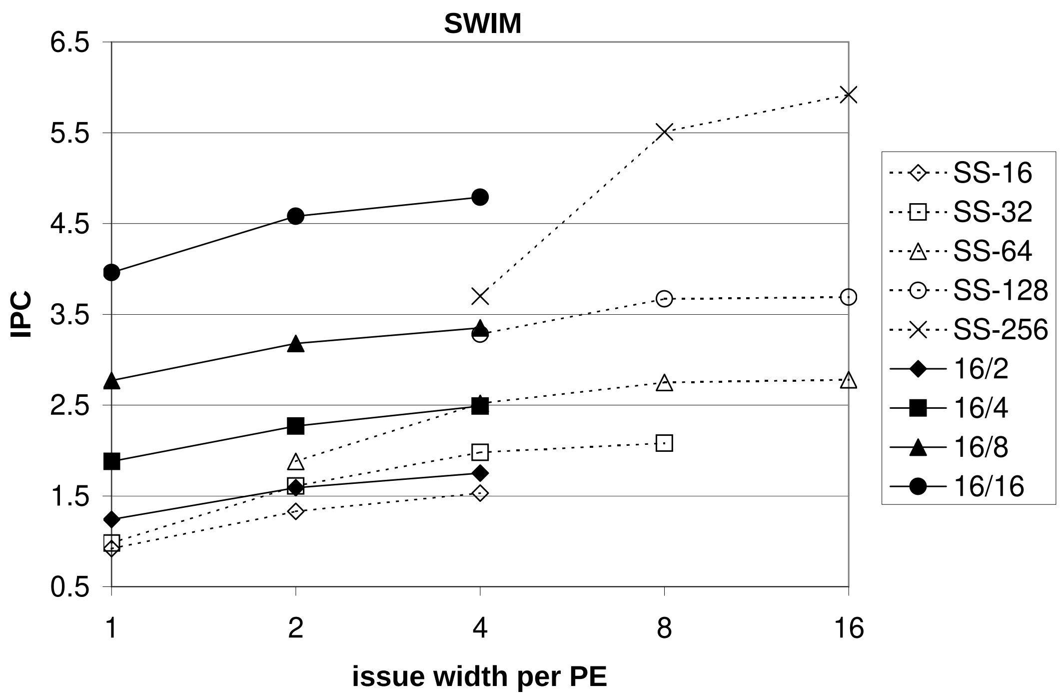


Figure 5-60: Superscalar vs. trace processors with partial bypasses (*swim*).

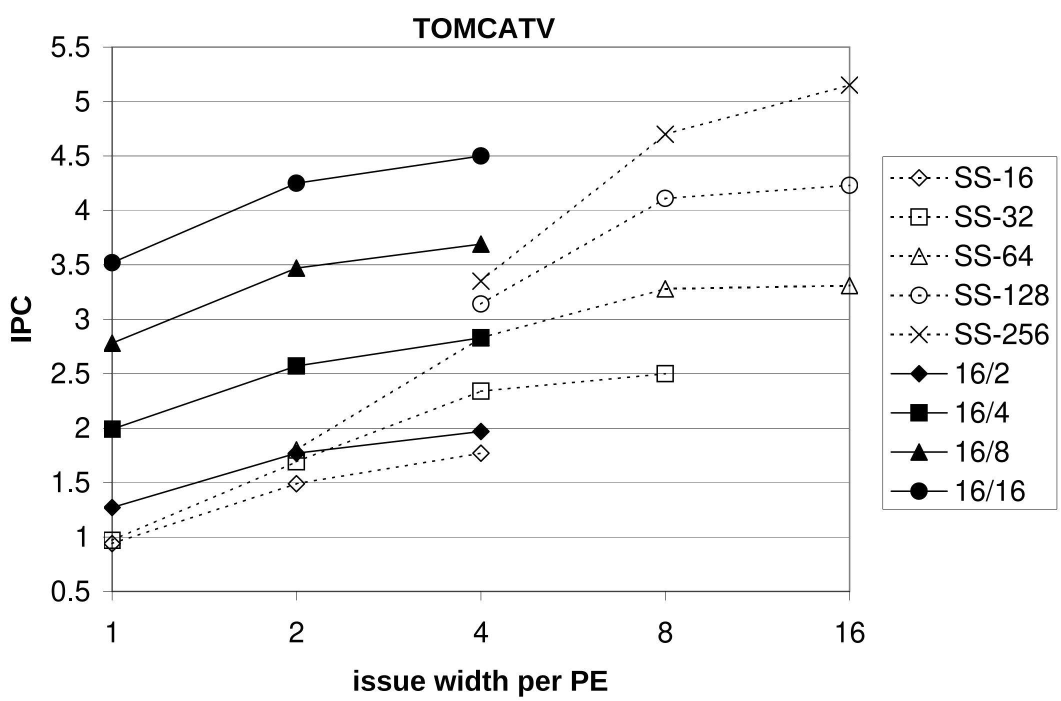


Figure 5-61: Superscalar vs. trace processors with partial bypasses (*tomcatv*).

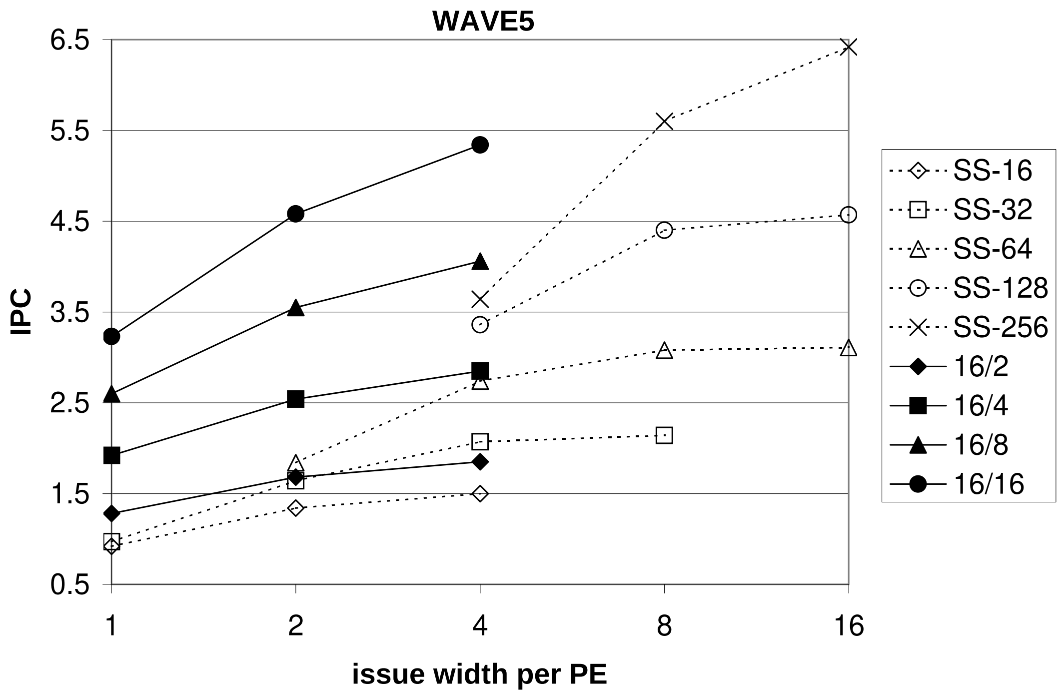


Figure 5-62: Superscalar vs. trace processors with partial bypasses (*wave5*).

5.6 Summary of hierarchy

5.6.1 High bandwidth instruction fetching

It is important to design instruction fetch units capable of fetching past multiple, possibly taken branches each cycle. Trace caches provide this capability without the complexity and latency of equivalent-bandwidth instruction cache designs [80]. The trace cache improves performance from 15% to 35% over an otherwise equally-sophisticated, but contiguous multiple-block fetch mechanism.

5.6.2 High bandwidth instruction execution

In most cases, the trace processor requires 2 or 4 times the *aggregate* issue bandwidth and/or window size to achieve the same IPC performance as a centralized (but otherwise equivalent) processor. This is due to 1) distribution effects such as *less flexible scheduling*, *window fragmentation*, and *discrete window management* (scheduling is the dominant factor in larger instruction windows) and 2) *partial operand bypasses* in the form of an extra cycle to communicate global register operands.

The relative complexity of a single PE is always less than that of a superscalar processor, however, and trace processor cycle time is more-or-less sensitive to single PE complexity. Stated another way, trace processor complexity increases relatively slowly as more PEs are added, whereas superscalar complexity is fairly sensitive to additional hardware parallelism -- especially additional issue bandwidth. Overall, *trace processors outperform aggressive superscalar counterparts because the trace processor microarchitecture enables both high ILP and a fast clock*.

Chapter 6

Evaluation of Speculation

This chapter evaluates the trace processor's advanced data and control speculation mechanisms. I begin with an analysis of data speculation techniques in Section 6.1: memory dependence speculation, live-in register prediction and speculation, and selective recovery. This is followed by an evaluation of the trace processor's novel control flow management techniques.

6.1 Data speculation

The trace processor performs two types of data speculation. Memory dependence speculation addresses the problem of ambiguous data dependences in large instruction windows. Live-in value prediction is the second form of data speculation. The relative importance of speculating inter-trace data dependences increases in trace processors because of the longer latency to communicate these values among PEs.

The following subsections evaluate the effectiveness of speculative disambiguation and live-in prediction. Implicit in this evaluation is the supporting data speculation mechanism, selective recovery. Measurements of the frequency and causes of selective re-issuing, and a comparison of selective and full squashing, are also provided.

6.1.1 Memory dependence speculation

Three memory disambiguation models are compared in this section: *oracle*, *spec*, and *non-spec*.

- *oracle*: This model implements oracle memory disambiguation. All load and store addresses are known prior to being computed, therefore, memory dependences are never ambiguous. *Oracle* represents an upper bound on memory disambiguation performance and is not a real hardware mechanism.
- *spec*: This is the trace processor's speculative memory disambiguation model, described in Section 3.3.2.
- *non-spec*: This model implements a conventional, non-speculative memory disambiguation mechanism. Loads wait for all prior store addresses to be computed and non-speculative. When this condition is met, if there are any dependences with prior stores, the dependent loads wait for the respective stores to complete (a store need only issue to a cache/ARB port to be considered complete).

The evaluation is limited in scope. Firstly, there are potentially other speculative mechanisms with which to compare. Some mechanisms may perform equally well without the need for selective recovery, in particular, mechanisms that explicitly predict memory dependences [63]. Secondly, modifications to my memory dependence speculation technique might obviate the need for selective recovery. For example, I have not investigated a combination of *non-spec* and *spec* in which stores issue to a cache/ARB port as soon as their addresses are available (even without data), causing dependent loads to synchronize and avoid disambiguation squashes. Nevertheless, the study does demonstrate the effectiveness of trace processor memory dependence speculation.

Figure 6-1 shows the performance of the three disambiguation models for `TP-128-32(16/8/4)`, with 4 global result buses, a global bypass latency of 1 cycle, and all other parameters configured as described in Section 4.2. *Oracle* performs 11% better than *non-spec*, on average. Disambiguation does not appear to be a serious problem for *compress*, *jpeg*, and *m88ksim*, since the performance difference between *oracle* and *non-spec* is no more than 4%. Of the remaining five benchmarks, four of them show a performance gain on the order of 10% to 15% with *oracle* disambiguation. Finally, disambiguation is a significant performance factor for *perl*, which shows a 34% performance difference between *non-spec* and *oracle* disambiguation.

The *spec* model performs nearly as well as *oracle* disambiguation. The IPC for *spec* is on average 2% lower than the IPC for *oracle*. Only in *m88ksim* does *spec* perform slightly worse than *non-spec*, but overall there is not a large IPC delta among the three models.

The small difference in performance between *spec* and *oracle* is due to 1) the one cycle penalty for re-issuing incorrectly speculated loads and 2) increased resource usage by incorrectly speculated loads and their dependent instructions. The one cycle penalty represents the latency to snoop a prior dependent store (misprediction detection latency).

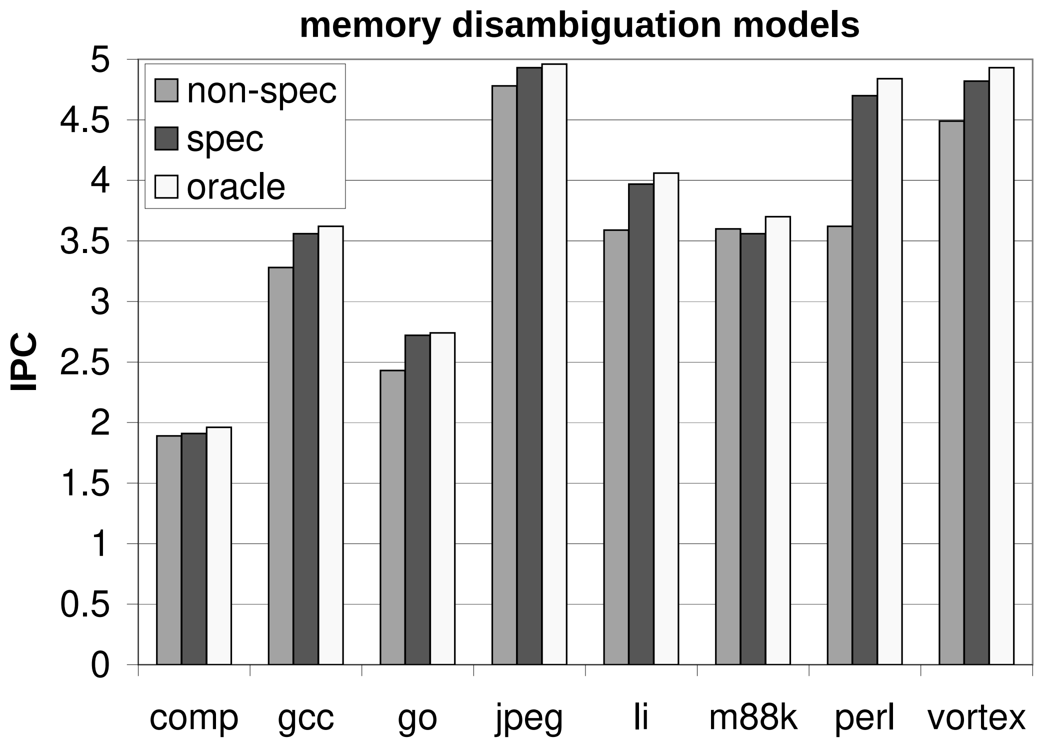


Figure 6-1: Performance of memory dependence speculation.

6.1.2 Live-in value prediction and speculation

Figures 6-2 through 6-8 shows trace processor performance as a function of global bypass latency, with no value prediction (*no VP*), context-based value prediction (*real VP*), and perfect value prediction (*perf VP*). The trace processor is `TP-128-16(16/8/2)` with 4 global result buses and all other parameters are configured as described in Section 4.2.

With *perf VP*, not only is IPC substantially improved, but the trace processor is much less sensitive to global bypass latency: the slope of the *perf VP* curve is closer to horizontal than either *no VP* or *real VP*. The *perf VP* processor is not entirely insensitive to global bypass latency because global values must still be communicated for 1) validating live-in predictions and 2) writing to the global register file.

A comparison of *no VP* and *real VP* reveals the following observations.

- For half of the benchmarks (*gcc*, *li*, *perl*, and *vortex*), live-in prediction recovers most of the performance lost due to the 1 cycle global bypass latency. That is, a processor with 1 cycle bypass and live-in prediction performs as well as a non-speculating processor with 0 cycle bypass. Likewise, a 2 cycle bypass with live-in prediction performs as well as 1 cycle bypass without live-in prediction, and so on.

  These benchmarks have live-in prediction accuracies between 50% and 75%, as shown in Figure 6-9 ("correct, confident").

- For one of the benchmarks, *m88ksim*, live-in value prediction does much more than recover the performance loss due to global communication latency. The live-in registers of *m88ksim* are correctly predicted nearly 90% of the time, as shown in Figure 6-9 ("correct, confident").

- Two of the benchmarks (*go* and *jpeg*) are virtually unaffected by live-in prediction. Live-in prediction accuracy is only about 25% for these benchmarks.

The first observation is encouraging. The trace processor reduces complexity by distributing resources, unfortunately at the expense of about 10% IPC due to global communication (Section 5.4.2). Value prediction recovers this performance loss.

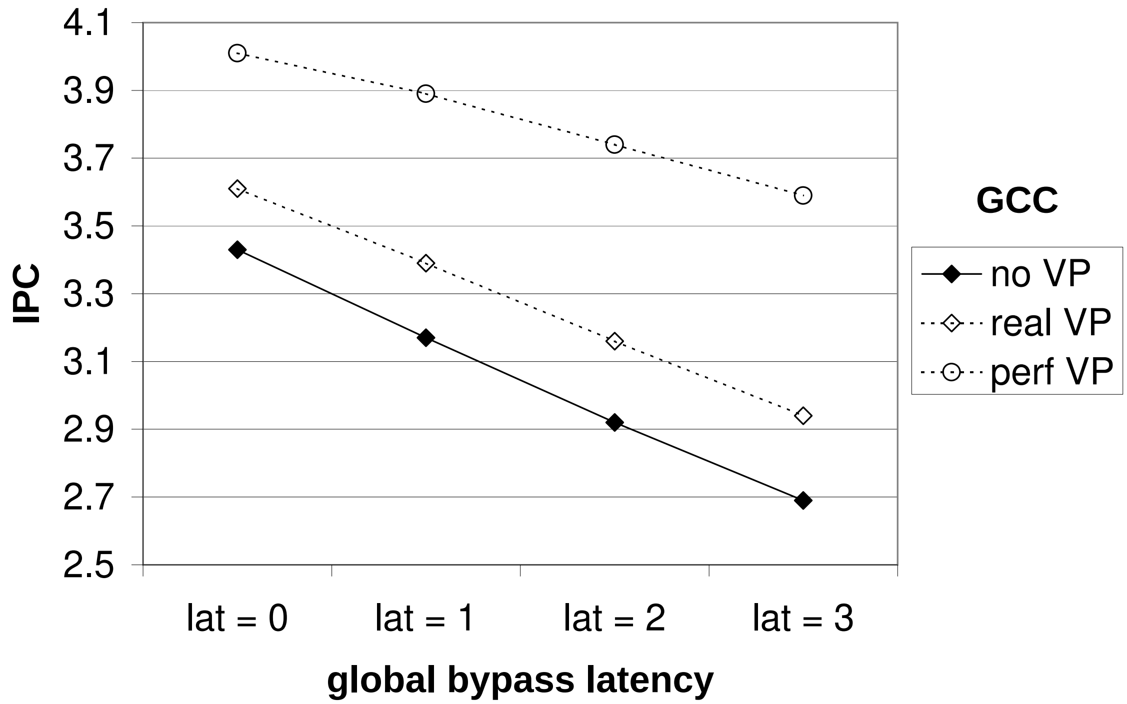


Figure 6-2: Live-in value prediction performance (*gcc*).

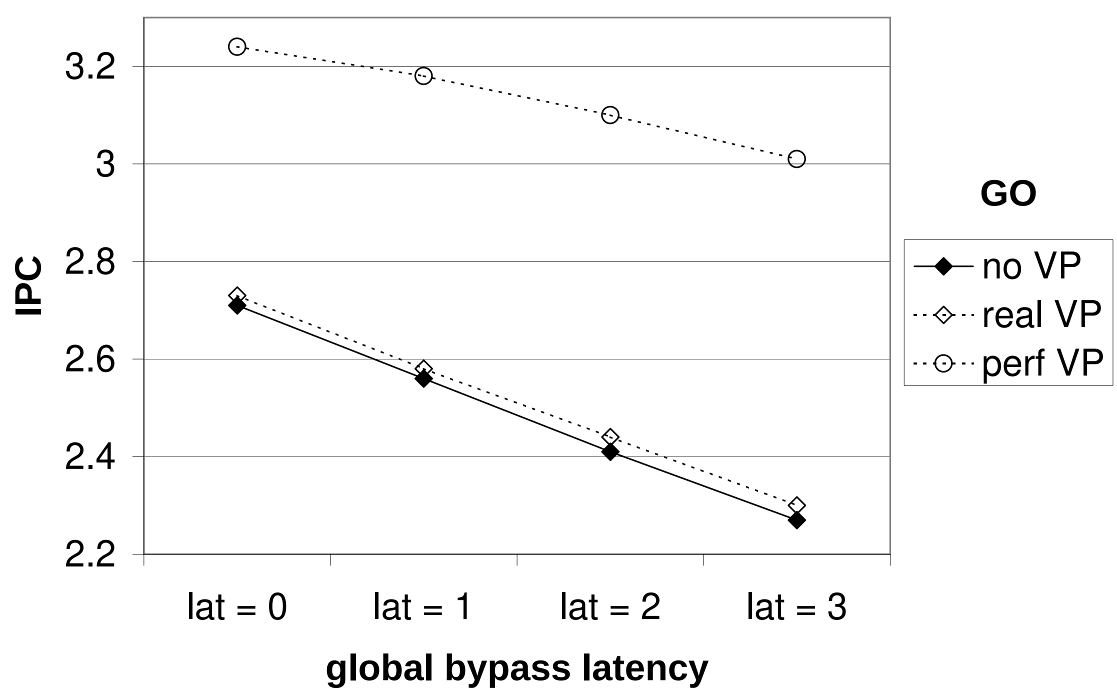


Figure 6-3: Live-in value prediction performance (*go*).

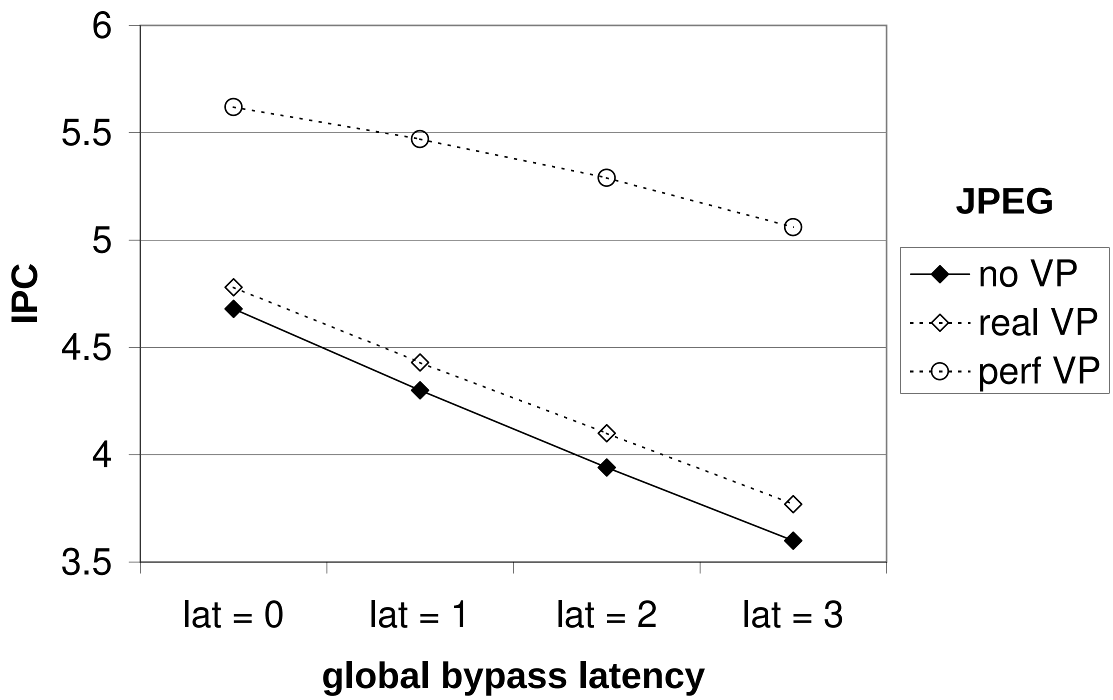


Figure 6-4: Live-in value prediction performance (*jpeg*).

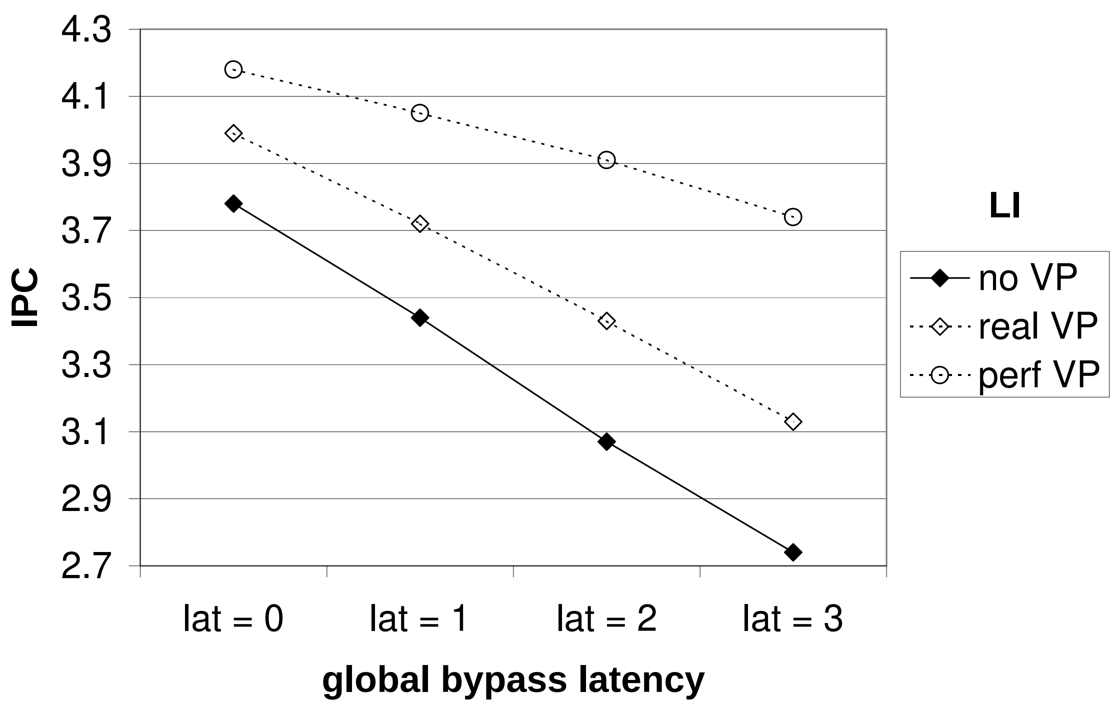


Figure 6-5: Live-in value prediction performance (*li*).

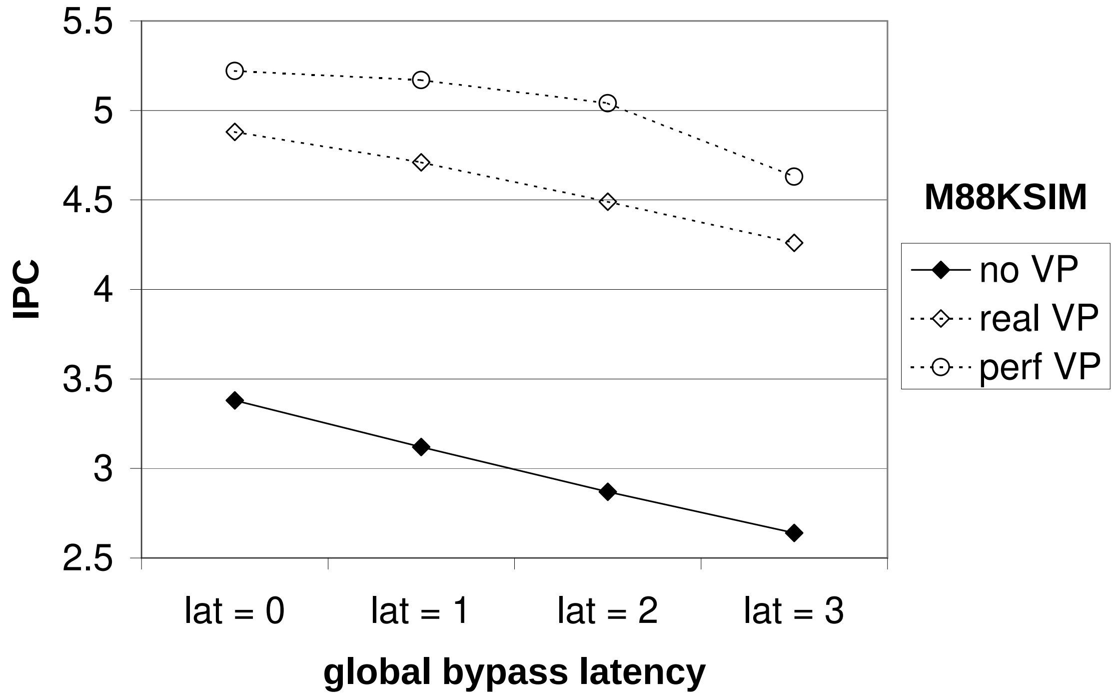


Figure 6-6: Live-in value prediction performance (*m88ksim*).

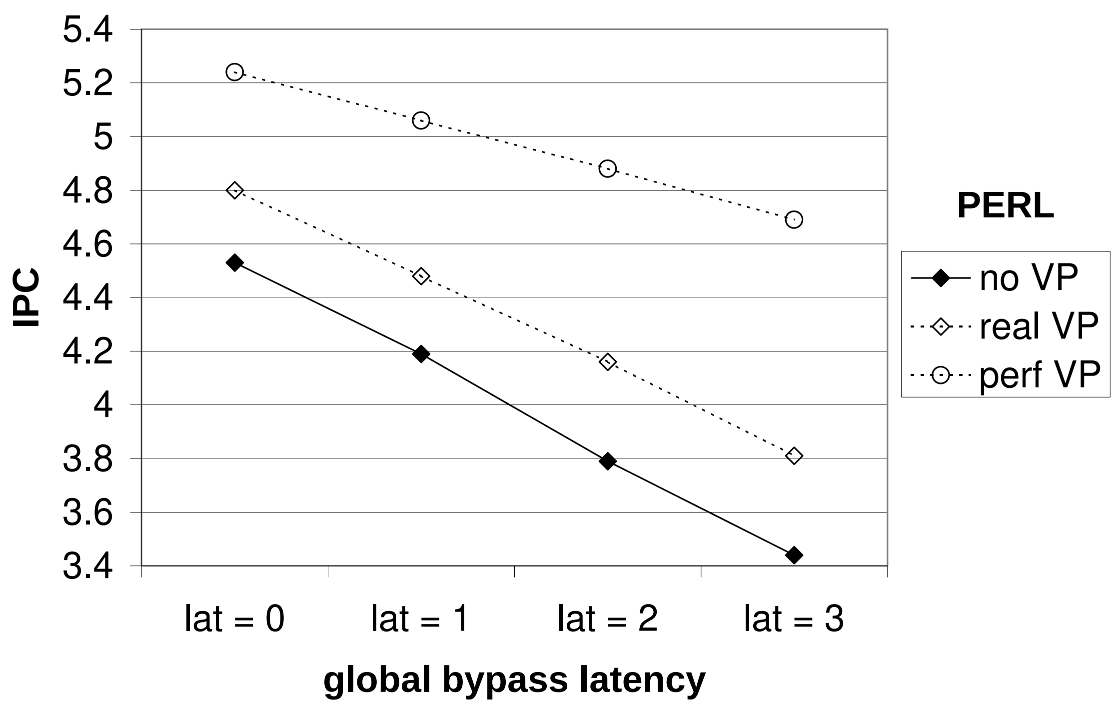


Figure 6-7: Live-in value prediction performance (*perl*).

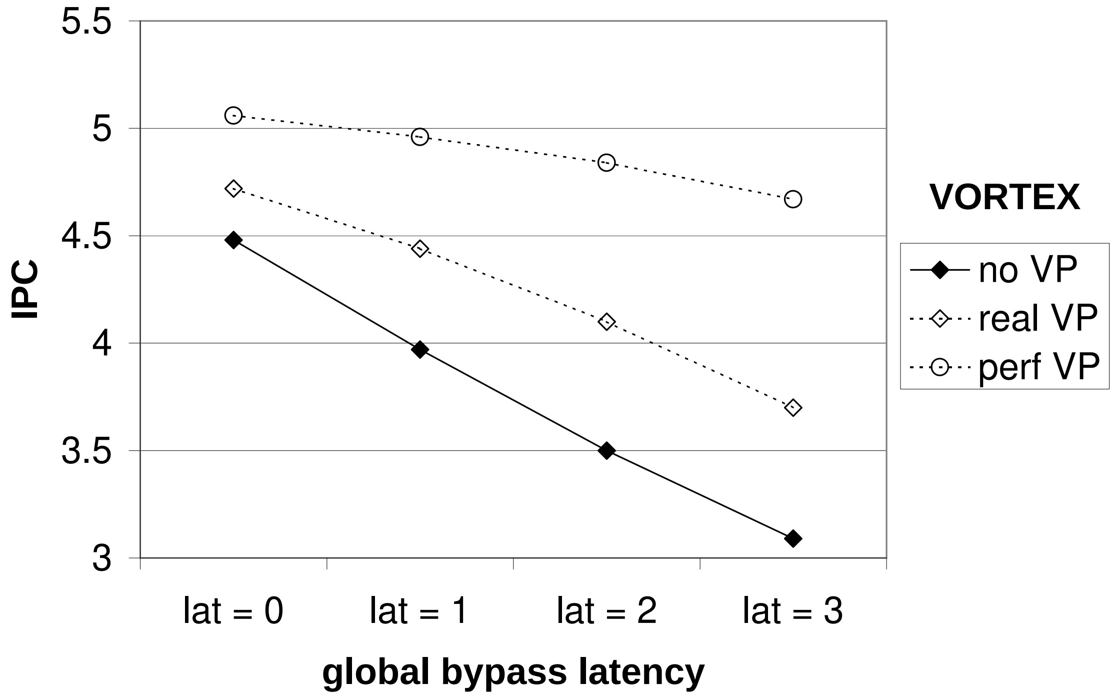


Figure 6-8: Live-in value prediction performance (*vortex*).

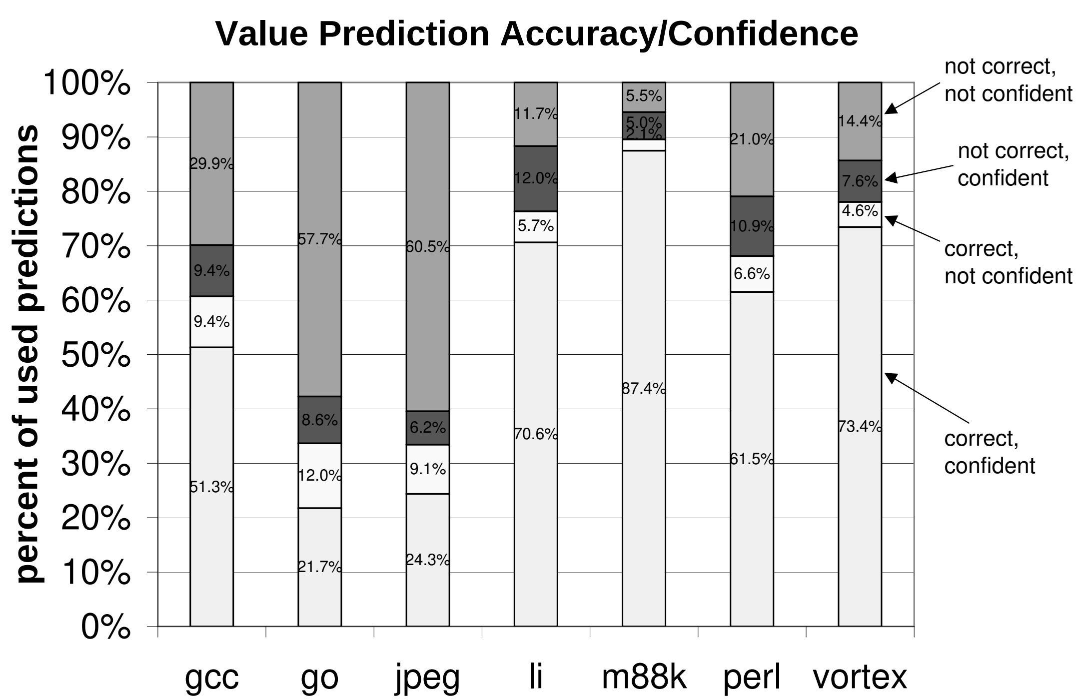


Figure 6-9: Live-in prediction accuracies.

6.1.3 Selective recovery

In this section, data misspeculation results are presented for the *real VP* model (defined in the previous section). Figure 6-10 shows the fraction of retired dynamic instructions that re-issue at least once. Three different graphs are shown. In the first graph, only memory dependence speculation is applied (“loads only”). In the second graph, only live-in prediction is applied (“live-ins only”), in which case *oracle* disambiguation is used. Finally, in the third graph, both types of data speculation are applied. Each graph shows the fraction of instructions that re-issue at least once. The bars are further broken down into 1) incorrectly disambiguated loads (labeled “disam”), i.e., loads that re-issue because they violated a memory dependence, 2) instructions that issued with an incorrect live-in prediction (labeled “live-in”), and 3) all other dependent instructions that re-issue (labeled “dep”).

Referring to the first and second graphs in Figure 6-10, overall, there does not appear to be much difference in re-issue rate with speculative loads or live-in prediction. In either graph, usually fewer than 2% of instructions are incorrectly disambiguated loads or live-in mispredictions, respectively, and usually fewer than 6% of all retired instructions re-issue. Referring to the third graph (“full data speculation”), on average, 10% of all retired instructions re-issue at least once. The majority are incorrect-data dependent instructions.

Figure 6-11 shows the number of times instructions issue while they are in the window (using full data speculation). Most instructions (85% - 93%) issue only once. A moderate fraction of instructions issue twice (5% - 12%). Instructions rarely issue three or more times.

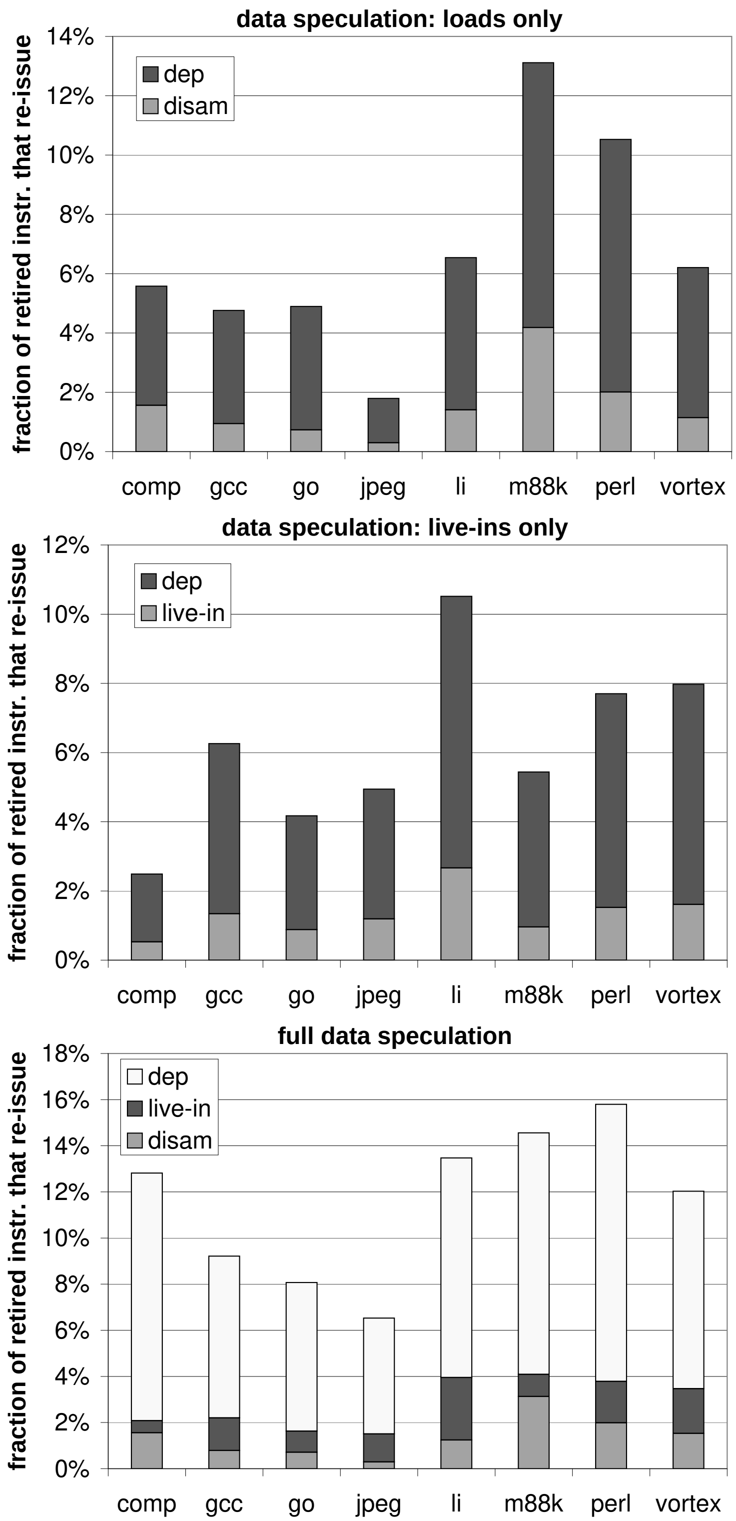


Figure 6-10: Fraction of retired dynamic instructions that issue more than once.

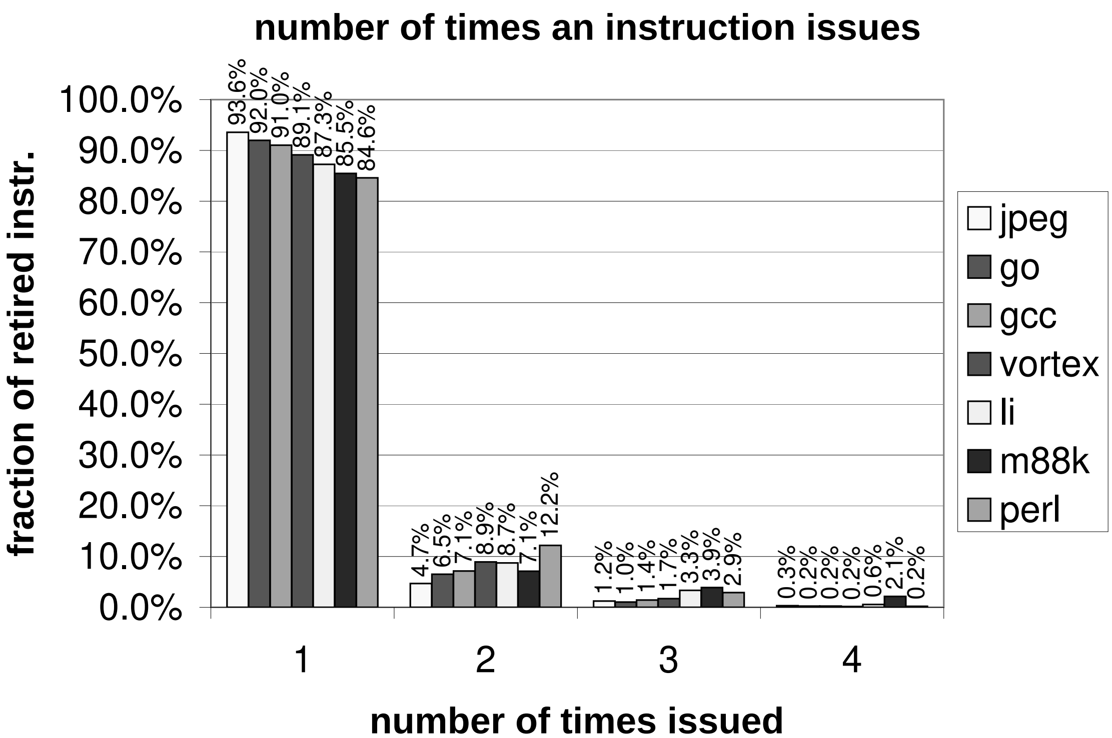


Figure 6-11: Number of times an instruction issues while in the window.

In Figure 6-12, I compare the performance of selective recovery with that of complete squashing due to data misspeculation. As before, three graphs are shown (load speculation only, live-in speculation only, and both). The performance degradation due to complete squashing is significant: an average decrease in performance of about 11% for load speculation, 7% for live-in speculation, and 16% for both. For all benchmarks except *jpeg*, selective recovery after load misspeculation is as or more important than selective recovery after live-in misspeculation.

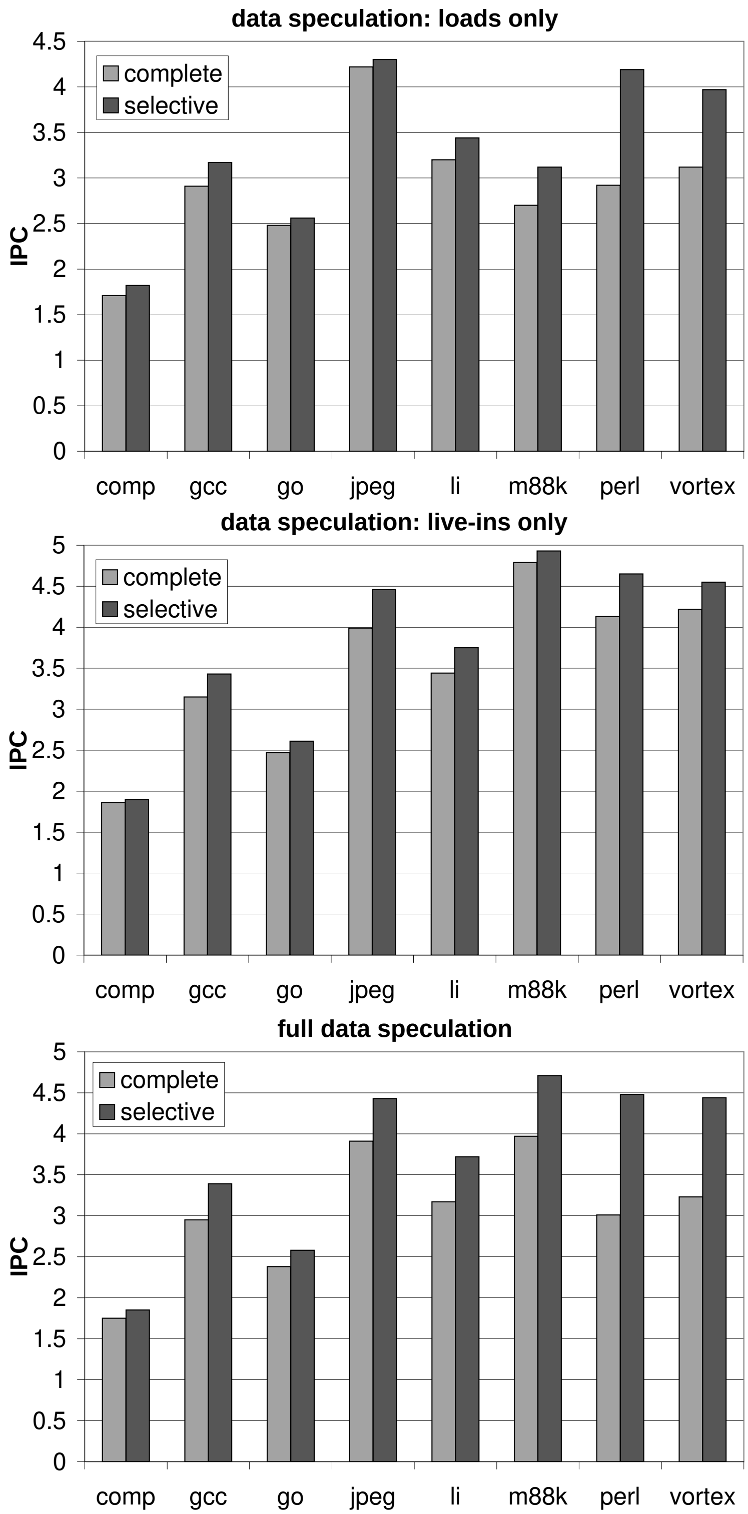


Figure 6-12: Selective versus complete squashing due to data misspeculation.

6.2 Control independence

Control independence is a promising technique for overcoming the branch misprediction bottleneck. Trace processors can exploit hierarchy to manage the complexity of implementing control independence. This section evaluates the effectiveness of trace processor control independence mechanisms in reducing branch misprediction penalties.

A trace processor with length-32 traces, 16 PEs, and 4-way issue per PE (i.e. `TP-512-64(32/16/4))` is simulated in anticipation of future large instruction windows, for which control independence techniques are likely to be more relevant. There are 8 global result buses and global bypass latency is 1 cycle. All other trace processor parameters are configured as described in Section 4.2.

Two sets of experiments are presented. The first set focuses on the impact of FGCI and CGCI *trace selection* in a trace processor without control independence mechanisms. This is required to isolate the effects of trace selection on trace cache performance, trace predictor performance, and PE utilization. The second set evaluates the performance of *control independence*.

6.2.1 Performance impact of trace selection

Default trace selection terminates traces at a maximum length of 32 instructions or at any jump indirect, call indirect, or return instruction. The *ntb* trace selection terminates traces at predicted not-taken backward branches, and *fg* denotes FGCI trace selection. The selection-only experiments are prefixed with *base* to indicate no control independence, and are followed by the trace selection algorithm(s) used. Default trace selection is always in effect and, therefore, is not explicitly specified. The four experiments are: *base*, *base(ntb)*, *base(fg)*, and *base(fg,ntb)*.

Performance results in instructions per cycle (IPC) are tabulated in Table 6-1 ("No Control Independence"). Also, the performance improvement with respect to *base* is graphed in Figure 6-13. Additional selection constraints (*fg*, *ntb*) tend to affect basic performance adversely. To help understand why, Table 6-2 shows the impact of selection on trace length, trace mispredictions, and trace cache misses (the latter two are given as misses per 1000 instructions and as a rate).

Additional selection constraints always decreases average trace length, and from Table 6-2, this almost always increases trace mispredictions per 1000 instructions. The trace predictor uses a path history of traces, and reducing the lengths of traces effectively reduces the amount of implicit branch history. Also, *synchronizing* trace selection among many disjoint paths -- in nested hammocks (*fg*) or after exiting a loop (*ntb*) -- reduces the number of unique traces significantly. Yet it is this uniqueness that provides a very distinct context for making predictions [84].

Reducing the average trace length also results in a waste of issue buffers in the PEs, effectively making the instruction window smaller. The only positive effect is on trace cache performance, but the benefit is generally overshadowed by costlier trace mispredictions.

The *base(ntb)* model improves performance slightly for three of the five benchmarks, but for *compress* and *li*, performance degrades by 5% and 10%, respectively. The effect is pronounced in *li* because trace length drops by 25%, double what other benchmarks experience. The *base(fg)* model degrades performance between 2% and 3% for four of five benchmarks.

Table 6-1: Performance in instructions per cycle (IPC).

	No Control Independence				Control Independence			
	base	base(ntb)	base(fg)	base(fg,ntb)	RET	MLB-RET	FG	FG + MLB-RET
gcc	4.44	4.51	4.34	4.36	4.68	4.78	4.51	4.73
go	3.17	3.20	3.07	3.10	3.73	3.81	3.15	3.65
comp	2.02	1.92	1.96	1.92	2.48	2.43	2.43	2.31
jpeg	7.12	7.24	6.96	6.96	7.21	7.33	8.79	8.89
li	4.72	4.31	4.72	4.34	5.23	4.83	4.79	4.84

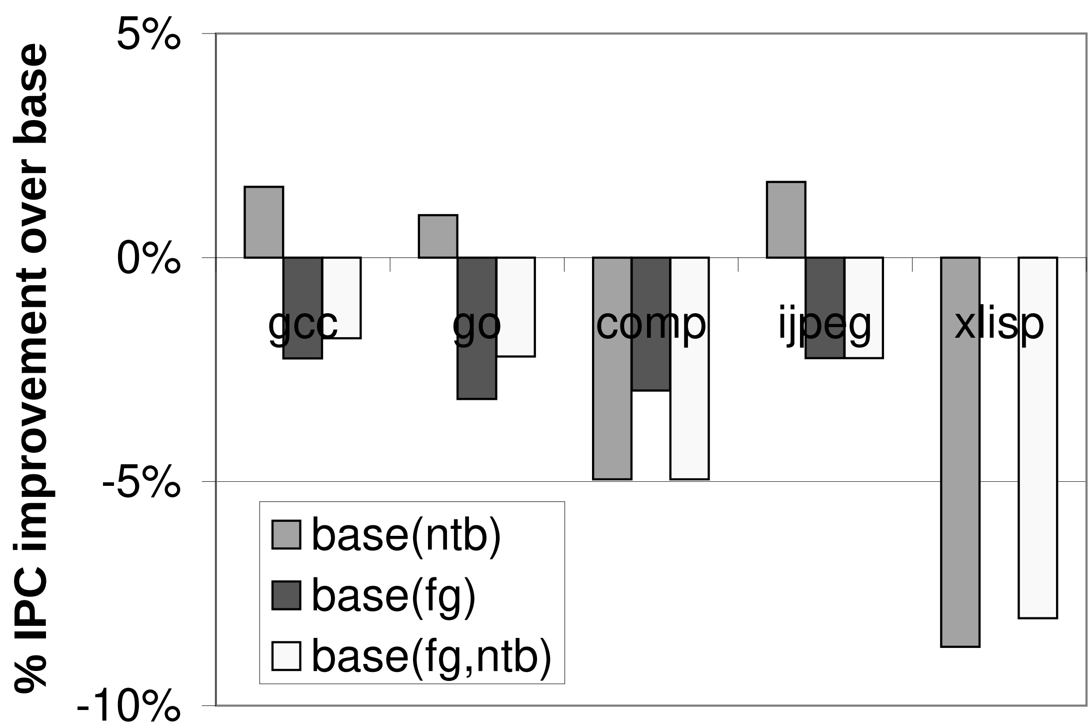


Figure 6-13: Performance impact of trace selection.

Table 6-2: Impact of trace selection on trace length, mispredictions, and misses.

		gcc	go	compress	jpeg	li
base	avg. trace length	24.0	27.2	24.9	31.1	19.7
	trace misp. rate	4.2 (10.1%)	7.3 (19.9%)	10.6 (26.3%)	3.1 (9.5%)	4.8 (9.4%)
	trace $ miss rate	4.7 (11.2%)	10.2 (27.7%)	0.0 (0.0%)	0.3 (1.1%)	0.0 (0.0%)
base(ntb)	avg. trace length	21.6	24.4	21.6	30.1	14.7
	trace misp. rate	4.3 (9.3%)	7.4 (18.1%)	11.2 (24.2%)	3.0 (9.0%)	6.0 (8.8%)
	trace $ miss rate	4.1 (8.8%)	9.7 (23.7%)	0.0 (0.0%)	0.3 (0.9%)	0.0 (0.0%)
base(fg)	avg. trace length	21.8	23.9	24.6	28.9	18.9
	trace misp. rate	4.4 (9.7%)	8.1 (19.2%)	10.8 (26.5%)	3.8 (11.0%)	4.9 (9.2%)
	trace $ miss rate	4.0 (8.8%)	9.4 (22.4%)	0.0 (0.0%)	0.2 (0.7%)	0.0 (0.0%)
base(fg,ntb)	avg. trace length	19.7	21.6	21.2	28.1	14.2
	trace misp. rate	4.7 (9.2%)	8.3 (17.9%)	10.9 (23.2%)	3.9 (10.8%)	6.0 (8.6%)
	trace $ miss rate	3.6 (7.2%)	9.0 (19.4%)	0.0 (0.0%)	0.2 (0.7%)	0.0 (0.0%)

6.2.2 Performance of control independence

In this section, I evaluate the performance of four control independence models:

- *RET*: coarse-grain only, using the *RET* heuristic.
- *MLB-RET*: coarse-grain only, using the *MLB-RET* heuristic.
- *FG*: fine-grain only.
- *FG + MLB-RET*: fine-grain and coarse-grain using the *MLB-RET* heuristic.

Table 6-1 (“Control Independence”) and Figure 6-14 (performance improvement over *base*) show that control independence improves performance substantially. Coarse-grain control independence performs uniformly well across the benchmarks, with the exception of *jpeg*. The *RET* model improves performance by about 5% for *gcc*, nearly 10% for *li*, and about 20% for *compress* and *go*. Going from *RET* to *MLB-RET* improves performance moderately for *gcc* and *go*, due to greater misprediction coverage and establishing more precise control independent points for backward branches. The improvement is perhaps moderate due to overlapping coverage between the two kinds of global re-convergent points. For *li*, *MLB-RET* drops performance with respect to *RET*, an artifact of *ntb* trace selection.

To give insight into the performance of FGCI, conditional branch statistics are shown in Table 6-3. Branches are classified into those that can be captured by FGCI, all other forward branches, and backward branches. FGCI branches are further divided into those whose regions fit (<=32) or do not fit (>32) in a trace; note that a trace length of 32 can capture almost all FGCI-type branches. The fraction of dynamic branches and mispredictions are given for each class.

FGCI branches account for 23% to 41% of all branches, and over 60% of all mispredictions, in *compress* and *jpeg*. This explains why the model *FG* performs very well on these benchmarks, namely a 20% to 25% performance improvement.

FGCI branches account for 24% of all mispredictions in *go*, yet *FG* actually degrades performance by less than 1%. Looking further into the misprediction behavior of *go*, I have noticed that FGCI has large potential in some frequently executed code (e.g. the “addlist” function), but that neighboring mispredictions not covered by FGCI nullify this potential; combined with the minor adverse effects of *fg* trace selection, the result is no gain. In contrast, the *MLB-RET* model performs well by capturing *clusters* of mispredictions in these same code regions.

Returning to Table 6-3, *gcc*, *go*, *jpeg,* and *li* have large FGCI regions (13 to 40 instructions) with many conditional branches enclosed (3 to 4). The average dynamic region size of FGCI branches is from 1 to 8 instructions smaller than the corresponding static code region.

Backward branches account for a large fraction of mispredictions, 20% for four of the benchmarks and 60% for *li*. Unfortunately for *li*, applying *ntb* trace selection -- so the *MLB-RET* heuristic can cover these mispredictions -- also worsens the prediction accuracy of the backward branches. While *MLB-RET* performs only slightly better than *base,* it improves performance by 10% over *base(ntb)* -- i.e. CGCI is being exploited, if only to break even with *base*.

In summary, using FGCI and CGCI techniques together achieves the best performance improvement on average: 13% (*FG + MLB-RET*). Clearly, some techniques work better than others depending on the benchmark, perhaps suggesting the need for adaptive trace selection. Using the best-performing technique for each benchmark, control independence achieves an average improvement of 17%.

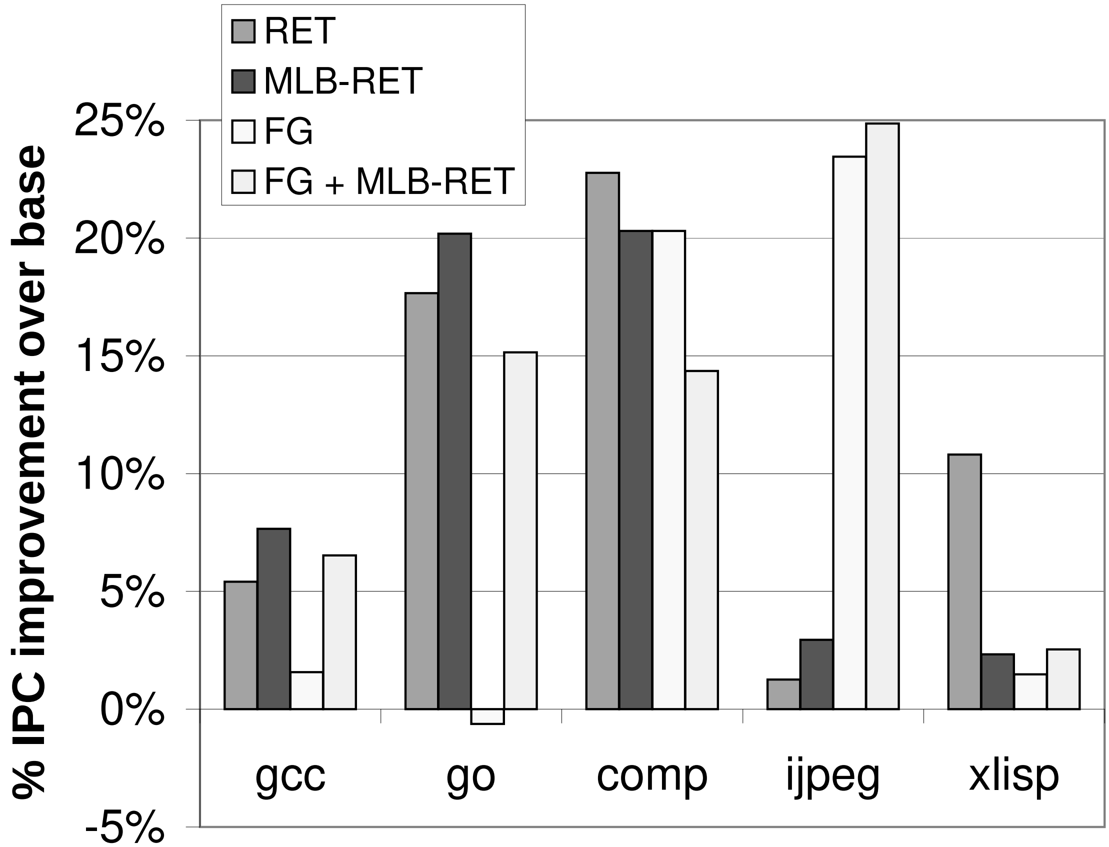


Figure 6-14: Performance of control independence.

Table 6-3: Conditional branch statistics.

			gcc	go	compress	jpeg	li
FGCI branches	<= 32	frac. br.	21.4%	24.5%	40.8%	22.5%	10.0%
		frac. misp.	20.3%	24.4%	63.1%	60.6%	3.0%
	> 32	frac. br.	1.9%	2.6%	0.1%	2.0%	0.0%
		frac. misp.	1.3%	2.7%	0.0%	1.9%	0.0%
		misp. rate	2.8%	8.7%	14.6%	14.8%	1.0%
		dyn. region size	11.3	13.8	4.3	31.9	13.2
		stat. region size	12.9	16.4	5.7	40.2	16.3
		# cond. br. in reg.	3.2	2.6	1.6	4.3	3.8
other forward branches		frac. br.	58.3%	52.8%	23.6%	24.8%	63.2%
		frac. misp.	55.8%	51.8%	17.8%	15.8%	36.1%
		misp. rate	2.9%	8.5%	7.1%	3.7%	1.9%
backward branches		frac. br.	18.4%	20.1%	35.5%	50.7%	26.7%
		frac. misp.	22.6%	21.1%	19.1%	21.7%	60.9%
		misp. rate	3.8%	9.1%	5.1%	2.5%	7.4%
overall branch misp. rate			3.1%	8.7%	9.4%	5.8%	3.3%
overall branch misp. per 1000 instr.			4.7	10.4	13.5	3.8	5.1

6.3 Summary of speculation

The trace processor uses data speculation to mitigate two types of data dependences. Ambiguous memory dependences impact performance moderately, on average about 10% -- the difference between conventional, conservative disambiguation and oracle disambiguation. Speculative disambiguation in a medium-sized trace processor removes nearly entirely the disambiguation problem. And live-in register prediction is introduced: the relative importance of speculating inter-trace data dependences increases in trace processors because of the longer latency to communicate these values among PEs. I simulated the effect of progressively clocking the PE faster by adding cycles to the global operand bypass latency, and made two observations. Value prediction in most cases does not significantly reduce the *sensitivity* to global bypass latency. But in many cases, the performance lost due to adding one extra bypass cycle is recovered by live-in prediction.

An aggressive, but relatively transparent, selective recovery model provides important data speculation support. About 10% of all retired instructions re-issue at least once during their lifetime in the window. “Sources” of misspeculation -- incorrectly disambiguated loads and instructions that issue with incorrect live-in predictions -- each represent only 10%-20% of all re-issued instructions, and the remaining 80%-90% are incorrect-data dependent instructions. On average, selectively re-issuing incorrect-data dependent instructions improves performance by nearly 20% over complete squashing.

Trace processors exploit control flow hierarchy and existing data speculation support (selective recovery) to manage the complexity of implementing control independence. The proposed control independence techniques improve trace processor performance by as much as 25%.

Chapter 7

Conclusion

In this thesis, I explored the trace processor microarchitecture, a processor organized entirely around *traces* -- dynamic instruction sequences, 16 to 32 instructions in length, which embed any number of taken or not-taken branch instructions. The trace processor searches far ahead into the program for instructions that may execute in parallel, thus exposing parallelism in apparently-serial programs. Constructing the large “instruction window” for exposing parallelism is normally complex, but trace processors use hierarchy to manage this complexity. Traces provide the hierarchy. Rather than work at the granularity of individual instructions, the processor efficiently sequences through the program at the higher level of traces and allocates trace-sized units of work to distributed processing elements (PEs). The trace-based approach overcomes basic inefficiencies of managing fetch and execution resources on an individual instruction basis.

The trace processor also uses aggressive speculation to partially alleviate the effects of data and control dependences. Dependences limit the amount of exploitable parallelism and, consequently, the peak bandwidth of the processor is often underutilized. Inter-trace data dependences are predicted to enhance parallel execution of traces among distributed PEs. Predicting inter-trace values is particularly relevant because these values are communicated globally, which can be slow. Ambiguous memory dependences are quickly resolved by issuing loads speculatively. The penalty of mispredicted branches (control dependences) is reduced by selectively preserving traces after a mispredicted branch that are control independent of the branch.

7.1 Overall results

I have demonstrated that the trace processor is a good microarchitecture for implementing wide-issue machines. Three key points support this conclusion.

1. *Trace processors are an evolutionary extension of superscalar processors*.

   Trace processors do not require instruction set architecture changes and, consequently, they maintain *binary compatibility*. Binary compatibility is arguably a major reason for the success of dynamic superscalar processors because it enables new processor generations to run existing software.

   Trace processors also retain a *single flow of control*. Wide instruction fetching enables instruction dependences to be established quickly and, likewise, instructions to be scheduled quickly. This approach is *robust* in that it performs well over a range of applications, e.g., applications with either irregular (integer programs) or regular (floating-point programs) parallelism. And it does not rely on sophisticated and potentially less-robust multithreading or VLIW compilers to accurately schedule instruction fetching from multiple, disjoint points in the program.

2. *Trace processors demonstrate better overall performance than conventional superscalar counterparts*.

   Distributing resources results in less scheduling flexibility and non-uniform operand bypassing, both of which reduce the average number of instructions executed per cycle. But the cycle time of a distributed processor is sensitive to *single PE complexity* and not the entire processor, which gives the trace processor an overall performance advantage over conventional superscalar processors. Overall, trace processors outperform aggressive superscalar counterparts because the trace processor microarchitecture enables both high ILP and a fast clock.

3. *The trace processor organization naturally supports aggressive speculation*.

   The contiguous instruction window enables aggressive, but relatively transparent, selective recovery from data misspeculation. Control flow hierarchy and existing data speculation support are leveraged to manage the complexity of exploiting control independence.

7.2 Detailed results

7.2.1 Trace-level sequencing

The trace processor frontend provides high instruction fetch bandwidth with low latency by predicting and caching sequences of multiple, possibly noncontiguous basic blocks. My evaluation of trace-level sequencing yielded the following results.

- The trace cache improves performance from 15% to 35% over an otherwise equally-sophisticated, but contiguous multiple-block fetch mechanism.
- Longer traces improve trace prediction accuracy. For misprediction-bound benchmarks, this factor contributes substantially to observed performance gains.
- A moderately large and associative trace cache performs as well as a perfect trace cache.
- With a robustly-designed core instruction fetch unit, overall performance is relatively insensitive to trace cache size and associativity. IPC varies no more than 10% over a wide range of trace cache configurations.
- An instruction cache combined with an aggressive trace predictor can fetch any number of contiguous basic blocks per cycle, yielding from 5% to 25% improvement over single-block fetching. This instruction supply model is incompatible with trace processors because traces are not cached and renamed as a unit. Nevertheless, it is an effective design point for conventional superscalar processors and requires only a 2-way interleaved instruction cache design.

7.2.2 Hierarchical instruction window

7.2.2.1 Distributing the instruction window

In most cases, the trace processor requires 2 or 4 times the *aggregate* issue bandwidth and/or window size to achieve the same IPC performance as a centralized (but otherwise equivalent) processor. This is due to 1) distribution effects such as *less flexible scheduling*, *window fragmentation*, and *discrete window management* (scheduling is the dominant factor in larger instruction windows) and 2) *partial operand bypasses* in the form of an extra cycle to communicate global register operands.

The relative complexity of a single PE is always less than that of a superscalar processor, however, and trace processor cycle time is more-or-less sensitive to single PE complexity. Stated another way, trace processor complexity increases relatively slowly as more PEs are added, whereas superscalar complexity is fairly sensitive to additional hardware parallelism -- especially additional issue bandwidth. Overall, *trace processors outperform aggressive superscalar counterparts because the trace processor microarchitecture enables both high ILP and a fast clock*.

7.2.2.2 Trace processor dimensions

I briefly explored the three trace processor dimensions, trace length, number of PEs, PE issue width, with the following results.

- For length-16 traces, it is important to provide 2-way out-of-order issue per PE. Beyond that, increasing the number of PEs is what derives the greatest performance benefit.
- For the same number of single-issue PEs, length-16 traces outperform length-32 traces, in spite of the smaller overall instruction window of the length-16 trace processor. Load imbalance is the primary cause, and doubling the per-PE issue bandwidth results in better performance with length-32 traces.

7.2.2.3 Hierarchical register model

The hierarchical division of registers is beneficial because fewer values are globally communicated. The global register file size and write ports are reduced, fewer global result buses are required, and fewer values incur the longer global bypass latency. In this thesis, I quantified the aforementioned benefits and showed them to be quite substantial, demonstrated that the benefits only increase with longer traces, and determined global parameters that perform as well as unconstrained global resources.

7.2.3 Data speculation

The trace processor uses data speculation to mitigate two types of data dependences. To handle *ambiguous* dependences, I developed a novel memory dependence solution based on the original address resolution buffer design. Speculative loads are tracked in the PEs instead of the ARB in order to facilitate selective re-issuing of misspeculated loads. Memory disambiguation has a moderate impact on performance, on average about 10% -- the difference between conventional, conservative disambiguation and oracle disambiguation. Speculative disambiguation in a medium-sized trace processor removes nearly entirely the disambiguation problem.

And live-in register prediction is introduced: the relative importance of speculating inter-trace data dependences increases in trace processors because of the longer latency to communicate these values among PEs. I simulated the effect of progressively clocking the PE faster by adding cycles to the global operand bypass latency, and made two observations. Value prediction in most cases does not significantly reduce the *sensitivity* to global bypass latency. But in many cases, the performance lost due to adding one extra bypass cycle is recovered by live-in prediction.

An aggressive, but relatively transparent, selective recovery model provides important data speculation support. Sources of data misspeculation are incorrectly speculated loads, incorrect live-in predictions, and incorrect-data dependent, control independent instructions. A variety of mispredictions may be active simultaneously and the interactions are complicated, but all scenarios are correctly handled.

7.2.4 Control independence

Control independence is a promising technique for mitigating the branch misprediction bottleneck. Trace processors exploit hierarchy to manage the complexity of implementing control independence, while maintaining the performance advantages of a contiguous instruction window and a relatively accurate single flow of control. My control independence ideas can be summarized as follows.

- A primary source of control independence complexity is the insertion and removal of instructions at arbitrary points in the window. Fortunately, the hierarchical instruction window of trace processors accommodates flexible control flow management. In the case of fine-grain control independence (FGCI), control flow recovery is localized within a single PE. In the case of coarse-grain control independence (CGCI), control flow recovery involves multiple PEs, but treating traces as the fundamental unit of control flow results in efficient recovery actions.

- Traces facilitate flexible control flow management but introduce a new problem: trace-level re-convergence is not guaranteed despite re-convergence at the instruction-level. Novel FGCI and CGCI trace selection techniques were developed for ensuring trace-level re-convergence.
- Trace processors exploit a variety of data speculation techniques and, therefore, already incorporate high-performance, selective data recovery mechanisms. These mechanisms are easily leveraged to selectively re-execute incorrect-data dependent, control independent instructions.

The proposed control independence techniques improve trace processor performance by as much as 25%.

7.3 Future research

The trace processor has many aspects and this thesis by no means covers them comprehensively. In the following subsections I describe future avenues of research.

7.3.1 Trace selection

Trace selection is a key future research topic because it impacts the qualities of traces and, therefore, all trace processor components: trace cache performance (tradeoffs between hit rate and bandwidth per hit); trace predictor behavior; trace construction latency; complexity (shifting complexity from PEs to global resources, and vice versa) and performance of the hierarchical instruction window; behavior of inter-trace dependences (both scheduling and prediction); and control independence performance. Trace line size in particular should be considered more carefully, and insightful measurements above and beyond IPC should be provided to understand underlying behavior.

7.3.2 Hybrid multiscalar and trace processor models

Another promising area of research is comparing and contrasting multiscalar and trace processors. Each has unique qualities that may improve the other. These qualities often stem from the basic differences between tasks and traces, therefore, understanding and articulating task/trace differences is an important future contribution. Then, the important qualities of each may be combined to provide a better overall hybrid processor.

For example, it may be beneficial to 1) have variable-size tasks and distributed fetch units in the PEs to locally construct tasks, and 2) predict all control flow through a task in a single cycle using a trace predictor. Variable size tasks may expose control independence better and increase the effective size of the window; local fetching achieves high fetch bandwidth without a trace cache (although multiple flows of control increases reliance on good task selection and intra-task scheduling); and predicting all control flow at once establishes data dependences among instructions quickly without having to actually fetch the instructions.

Another example is decoupling trace line size and PE window size. In this way, longer traces may be used for improving various aspects of the trace processor without impacting the processing element. This approach may complicate selective recovery, however.

7.3.3 Control independence

Much remains to be explored and explained of trace processor control independence. Important intermediate results and insights have been intentionally left out of the thesis because the concepts are too many and too unrefined. Yet these insights are important for understanding the behavior of control independence and improving the techniques for exploiting it.

For example, exploration of other, more sophisticated CGCI heuristics holds the potential for larger performance gains. Introducing other types of global re-convergent points may increase control independence opportunity. The impact on trace length must be minimized, however, since trace predictor performance and PE utilization are often adversely affected by additional trace selection constraints. Further, with more frequent trace-level re-convergent points in the instruction stream, more accurate heuristics will be needed for pairing mispredicted branches with re-convergent points.

7.3.4 Live-in prediction

Live-in prediction is potentially a key *enabler* for distributing the instruction window, clocking the chip very fast, and necessarily lengthening global latencies. But this aspect is largely uninvestigated here. Because global communication is likely to become costlier, trace selection that exposes predictable inter-trace dependences is an interesting area of future research.

7.3.5 Implementation studies

Low-level trace processor design issues have yet to be critically studied. A particularly interesting area is advanced control and data speculation -- for example, tracking speculative loads and detecting misspeculated loads, selective re-issuing, detailed control independence mechanisms, etc.

7.3.6 SMT in trace processors

Simultaneous multithreading (SMT) [108,109] is an important emerging microarchitectural technique. The basic idea is to run multiple threads (either from the same program or different programs) on a processor *at the same time*. SMT leverages the fine-grain scheduling flexibility and highly-parallel microarchitecture of wide-issue superscalar processors. Often, there are phases of a single program that do not fully utilize the microarchitecture, so sharing the processor resources among multiple programs will increase overall utilization. Improved utilization reduces the total time required to execute all program threads, despite possibly slowing down single thread performance.

The two major advantages of SMT are as follows.

1. The fact that there are multiple, independent threads co-existing in the scheduling window is rather transparent to the dynamic scheduler because existing register renaming techniques properly manage data dependences.

2. SMT manages resources at a *fine granularity* and on a *cycle-by-cycle* basis. Scheduling is extremely flexible and dynamic. In a given cycle, if a single program has a large amount of instructions that may issue in parallel, that one program may use the peak

bandwidth. In the next cycle, the same bandwidth may be partitioned equally among many threads.

The first point is important because it means SMT can be implemented in a fairly straightforward manner, and in a way that leverages existing superscalar mechanisms. The second point is important for two reasons. First, it suggests higher overall throughput is possible than if a coarse-grain, symmetric multiprocessor (SMP) machine were used. Second, it provides the impetus for building wider processors because a wide core does not have to rely solely on instruction-level parallelism to be viable.

For all of these reasons, SMT is likely to be implemented in future microprocessors. Therefore, it is important to think about SMT in trace processors. An obvious approach is to time-share the trace processor frontend among threads (i.e. only one thread uses the trace predictor/trace cache in a given cycle) and space-share the PEs among threads (each thread occupies a certain number of PEs) [85]. There are several benefits to the trace processor SMT approach. First, instruction fetch bandwidth has been identified as a major bottleneck in SMT machines [109], and the trace cache naturally provides high fetch bandwidth. Second, it is conceptually simpler to manage shared resources at the trace-level (PEs) than at the instruction-level. In other words., all of the benefits of hierarchy (both instruction fetching and execution) transfer to SMT.

But there are two possible problem areas. The trace cache requires more storage than a conventional instruction cache due to redundancy; this could become especially problematic if multiple threads share the trace cache (e.g., very high miss rates). Also, by coarsely space-sharing execution resources, the SMT advantage of fine-grain scheduling flexibility is potentially diluted. Clearly, SMT in trace processors is an interesting future research topic.

Bibliography

[1] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath Execution: Opportunities and Limits. *International Conference on Supercomputing*, July 1998.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. *31st International Symposium on Microarchitecture*, December 1998.

[3] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. *10th Symposium on Principles of Programming Languages*, January 1983.

[4] H. Ando, C. Nakanishi, T. Hara, and M. Nakaya. Unconstrained Speculative Execution with Predicated State Buffering. *22nd International Symposium on Computer Architecture*, June 1995.

[5] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. *ACM Conference on Programming Language Design and Implementation*, June 1991.

[6] M. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. In *1995 International Electron Devices Meeting Technical Digest*, pages 241-244, 1995.

[7] J. Bondi, A. Nanda, and S. Dutta. Integrating a Misprediction Recovery Cache (MRC) into a Superscalar Pipeline. *29th International Symposium on Microarchitecture*, pages 14–23, December 1996.

[8] S. Breach, T. Vijaykumar, and G. Sohi. The Anatomy of the Register File in a Multiscalar Processor. *27th International Symposium on Microarchitecture*, pages 181–190, November 1994.

[9] S. Breach, T. Vijaykumar, S. Gopal, J. Smith, and G. Sohi. Data Memory Alternatives for Multiscalar Processors. Technical Report CS-TR-97-1344, Computer Sciences Department, University of Wisconsin - Madison, November 1996.

[10] S. Breach. *Design and Evaluation of a Multiscalar Processor*. PhD Thesis, University of Wisconsin - Madison, February 1999.

[11] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.

[12] Y. Chou, J. Fung, and J. Shen. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. *International Conference on Supercomputing*, June 1999.

[13] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. *22nd International Symposium on Computer Architecture*, pages 333–344, June 1995.

[14] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, pages 536–538.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An Efficient Method of Computing Static Single Assignment Form. *ACM Symposium on Principles of Programming Languages*, January 1989.

[16] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-grained Multithreading. *International Conference on Parallel Architecture and Compilation Techniques*, 1995.

[17] S. Dutta and M. Franklin. Control Flow Prediction with Tree-like Subgraphs for Superscalar Processors. *28th International Symposium on Microarchitecture*, pages 258–263, November 1995.

[18] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. *30th International Symposium on Microarchitecture*, pages 149-159, December 1997.

[19] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[20] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-grain Parallelism. *19th International Symposium on Computer Architecture*, May 1992.

[21] M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. *25th International Symposium on Microarchitecture*, November 1992.

[22] M. Franklin. *The Multiscalar Architecture*. PhD Thesis, University of Wisconsin - Madison, November 1993.

[23] M. Franklin and M. Smotherman. A Fill-unit Approach to Multiple Instruction Issue. *27th International Symposium on Microarchitecture*, pages 162–171, November 1994.

[24] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[25] D. Friendly, S. Patel, and Y. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. *30th International Symposium on Microarchitecture*, pages 24–33, December 1997.

[26] D. Friendly, S. Patel, and Y. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. *31st International Symposium on Microarchitecture*, pages 173–181, December 1998.

[27] F. Gabbay and A. Mendelson. Speculative Execution Based on Value Prediction. Technical Report 1080, Electrical Engineering Department, Technion - Israel Institute of Technology, November 1996.

[28] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. *4th International Symposium on High-Performance Computer Architecture*, February 1998.

[29] G. F. Grohoski. Machine Organization of the IBM RS/6000 Processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[30] L. Gwennap. MIPS R10000 Uses Decoupled Architecture. *Microprocessor Report*, October 1994.

[31] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer* (Special Issue on Billion-Transistor Processors), September 1997.

[32] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. *8th International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[33] E. Hao, P.-Y. Chang, M. Evers, and Y. Patt. Increasing the Instruction Fetch Rate via Block-structured Instruction Set Architectures. *29th International Symposium on Microarchitecture*, pages 191–200, December 1996.

[34] T. Heil and J. Smith. Selective Dual Path Execution. Technical Report, Department of Electrical and Computer Engineering, University of Wisconsin - Madison, November 1996.

[35] W. Hwu and P. Chang. Trace Selection for Compiling Large C Application Programs to Microcode. *21st International Symposium on Microarchitecture*, December 1988.

[36] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, January 1993.

[37] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. *29th International Symposium on Microarchitecture*, pages 142–152, December 1996.

[38] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control Flow Speculation in Multiscalar Processors. *3rd International Symposium on High-Performance Computer Architecture*, pages 218–229, February 1997.

[39] Q. Jacobson, E. Rotenberg, and J. Smith. Path-Based Next Trace Prediction. *30th International Symposium on Microarchitecture*, pages 14–23, December 1997.

[40] Q. Jacobson and J. Smith. Instruction Pre-Processing in Trace Processors. *5th International Symposium on High-Performance Computer Architecture*, January 1999.

[41] J. Johnson. Expansion Caches for Superscalar Processors. Technical Report CSL-TR-94-630, Computer Science Laboratory, Stanford University, June 1994.

[42] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. *9th Microprocessor Forum*, October 1996.

[43] G. Kemp and M. Franklin. Pews: A Decentralized Dynamic Scheduler for ILP Processing. *International Conference on Parallel Processing*, pages 239-246, 1996.

[44] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the Polypath Architecture. *25th International Symposium on Computer Architecture*, June 1998.

[45] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. *International Conference on Parallel Architecture and Compilation Techniques*, October 1998.

[46] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. *19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[47] J. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 21(7):6–22, January 1984.

[48] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. *Proceedings Compcon*, pages 28-36, 1997.

[49] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[50] M. Lipasti and J. Shen. Exceeding the Dataflow Limit via Value Prediction. *29th International Symposium on Microarchitecture*, December 1996.

[51] M. Lipasti. *Value Locality and Speculative Execution*. PhD Thesis, Carnegie Mellon University, April 1997.

[52] M. Lipasti and J. Shen. Superspeculative Microarchitecture for Beyond AD 2000. *IEEE Computer* (Special Issue on Billion-Transistor Processors), September 1997.

[53] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *22nd International Symposium on Computer Architecture*, June 1995.

[54] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. *International Conference on Supercomputing*, June 1999.

[55] P. Marcuello and A. González. Exploiting Speculative Thread-Level Parallelism on a SMT Processor. *International Conference on High Performance Computing and Networking*, April 1999.

[56] P. Marcuello and A. González. Speculative Multithreaded Processors. *International Conference on Supercomputing*, 1998.

[57] S. McFarling. Combining Branch Predictors. Technical Report TN-36, WRL, June 1993.

[58] S. Melvin, M. Shebanow, and Y. Patt. Hardware Support for Large Atomic Units in Dynamically Scheduled Machines. *21st International Symposium on Microarchitecture*, pages 60–66, December 1988.

[59] S. Melvin and Y. Patt. Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines. *3rd International Conference on Supercomputing*, pages 427–432, June 1989.

[60] S. Melvin and Y. Patt. Exploiting Fine-grained Parallelism through a Combination of Hardware and Software Techniques. *18th International Symposium on Computer Architecture*, pages 287–296, May 1991.

[61] S. Melvin and Y. Patt. Enhancing Instruction Scheduling with a Block-structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, 1995.

[62] T. Mitra. *Performance Evaluation of Improved Superscalar Issue Mechanisms*. Masters Thesis, Indian Institute of Science - Bangalore, January 1997.

[63] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. *24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[64] R. Nair and M. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. *24th International Symposium on Computer Architecture*, pages 13–25, June 1997.

[65] B. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. *23rd International Symposium on Computer Architecture*, May 1996.

[66] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[67] K. Olukotun, L. Hammond, and M. Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. *International Conference on Supercomputing*, June 1999.

[68] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors. Technical Report CSL-TR-97-715, Computer Systems Laboratory, Stanford University, February 1997.

[69] S. Palacharla, N. Jouppi, and J. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, Computer Sciences Department, University of Wisconsin - Madison, November 1996.

[70] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. *24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[71] S. Patel, D. Friendly, and Y. Patt. Critical Issues Regarding the Trace Cache Fetch Mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan - Ann Arbor, 1997.

[72] S. Patel, M. Evers, and Y. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. *25th International Symposium on Computer Architecture*, pages 262–271, June 1998.

[73] S. Patel, D. Friendly, and Y. Patt. Evaluation of Design Options for the Trace Cache Fetch Mechanism. *IEEE Transactions on Computers* (special issue on cache memory), 48(2):193–204, February 1999.

[74] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer* (Special Issue on Billion-Transistor Processors), September 1997.

[75] A. Peleg and U. Weiser. Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line. U.S. Patent Number 5,381,533, January 1995.

[76] D. Pnevmatikatos and G. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. *21st International Symposium on Computer Architecture*, April 1994.

[77] N. Ranganathan and M. Franklin. An Empirical Study of Decentralized ILP Execution Models. *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272-281, October 1998.

[78] N. Ranganathan and M. Franklin. Complexity-Effective PEWs Microarchitecture. *Microprocessors and Microsystems*, 1998.

[79] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. Technical Report CS-TR-96-1310, Computer Sciences Department, University of Wisconsin - Madison, April 1996.

[80] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *29th International Symposium on Microarchitecture*, pages 24–34, December 1996.

[81] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. *30th International Symposium on Microarchitecture*, pages 138–148, December 1997.

[82] E. Rotenberg, Q. Jacobson, and J. Smith. A Study of Control Independence in Superscalar Processors. Technical Report CS-TR-98-1389, Computer Sciences Department, University of Wisconsin - Madison, December 1998.

[83] E. Rotenberg, Q. Jacobson, and J. Smith. A Study of Control Independence in Superscalar Processors. *5th International Symposium on High-Performance Computer Architecture*, January 1999.

[84] E. Rotenberg, S. Bennett, and J. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers* (special issue on cache memory), 48(2):111–120, February 1999.

[85] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *29th Fault-Tolerant Computing Symposium*, June 1999.

[86] E. Rotenberg and J. Smith. Control Independence in Trace Processors. *32nd International Symposium on Microarchitecture*, November 1999.

[87] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. *29th International Symposium on Microarchitecture*, pages 238–247, December 1996.

[88] Y. Sazeides and J. Smith. The Predictability of Data Values. *30th International Symposium on Microarchitecture*, December 1997.

[89] Y. Sazeides. *An Analysis of Value Predictability and its Application to a Superscalar Processor*. PhD Thesis, University of Wisconsin - Madison, 1999.

[90] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors, 1994.

[91] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block Ahead Branch Predictors. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[92] J. E. Smith. A Study of Branch Prediction Strategies. *8th Symposium on Computer Architecture*, pages 135–148, May 1981.

[93] J. Smith and A. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, vol. 37, pages 562-573, May 1988.

[94] J. Smith and G. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609–24, December 1995.

[95] J. Smith. Amdahl's Law: Not Just an Equation. Keynote talk at the *International Symposium on Computer Architecture*, June 1997.

[96] J. Smith and S. Vajapeyam. Trace Processors: Moving to Fourth-Generation Microarchitectures. *IEEE Computer* (Special Issue on Billion-Transistor Processors), September 1997.

[97] M. Smotherman and M. Franklin. Improving CISC Instruction Decoding Performance Using a Fill Unit. *28th International Symposium on Microarchitecture*, pages 219–229, November 1995.

[98] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. *24th International Symposium on Computer Architecture*, June 1997.

[99] G. Sohi. Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, vol. 39, pages 349-359, March 1990.

[100] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[101] E. Sprangle and Y. Patt. Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme. *27th International Symposium on Microarchitecture*, pages 143–147, December 1994.

[102] J. Steffan and T. Mowry. The Potential for Using Thread-level Data Speculation to Facilitate Automatic Parallelization. *4th International Symposium on High-Performance Computer Architecture*, February 1998.

[103] K. Sundararaman and M. Franklin. Multiscalar Execution along a Single Flow of Control. *International Conference on Parallel Processing*, August 1997.

[104] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of Loops with Exits on Pipelined Architectures. *Supercomputing '90*, November 1990.

[105] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[106] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation. *International Conference on Parallel Architecture and Compilation Techniques*, 1996.

[107] J. Tubella and A. González. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. *4th International Symposium on High-Performance Computer Architecture*, February 1998.

[108] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *22nd International Symposium on Computer Architecture*, pages 392-403, June 1995.

[109] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *23rd International Symposium on Computer Architecture*, pages 191-202, May 1996.

[110] G. Tyson, K. Lick, and M. Farrens. Limited Dual Path Execution. Technical Report CSE-TR-346-97, Department of Electrical Engineering and Computer Science, University of Michigan - Ann Arbor, 1997.

[111] A. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. *28th International Symposium on Microarchitecture*, December 1995.

[112] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. *24th International Symposium on Computer Architecture*, pages 1–12, June 1997.

[113] T. N. Vijaykumar, S. E. Breach, and G. S. Sohi. Register Communication Strategies for the Multiscalar Architecture. Technical Report 1333, Computer Sciences Department, University of Wisconsin - Madison, February 1997.

[114] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD Thesis, University of Wisconsin - Madison, January 1998.

[115] T. N. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. *31st International Symposium on Microarchitecture*, December 1998.

[116] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. *25th International Symposium on Computer Architecture*, June 1998.

[117] T.-Y. Yeh and Y. N. Patt. Alternative Implementations of Two-level Adaptive Branch Prediction. *19th International Symposium on Computer Architecture*, May 1992.

[118] T.-Y. Yeh and Y. N. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *25th International Symposium on Microarchitecture*, pages 129–139, December 1992.

[119] T.-Y. Yeh. *Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors*. PhD Thesis, University of Michigan - Ann Arbor, 1993.

[120] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. *7th International Conference on Supercomputing*, pages 67–76, July 1993.