Optimization of Instruction Fetch Mechanisms for High Issue Rates

Thomas M. Conte Kishore N. Menezes Patrick M. Mills Burzin A. Patel
Computer Architecture Research Laboratory
Department of Electrical and Computer Engineering
University of South Carolina
Columbia, South Carolina

Abstract

Recent superscalar processors issue four instructions per cycle. These processors are also powered by highly-parallel superscalar cores. The potential performance can only be exploited when fed by high instruction bandwidth. This task is the responsibility of the instruction fetch unit. Accurate branch prediction and low I-cache miss ratios are essential for the efficient operation of the fetch unit. Several studies on cache design and branch prediction address this problem. However, these techniques are not sufficient. Even in the presence of efficient cache designs and branch prediction, the fetch unit must continuously extract multiple, non-sequential instructions from the instruction cache, realign these in the proper order, and supply them to the decoder. This paper explores solutions to this problem and presents several schemes with varying degrees of performance and cost. The most-general scheme, the collapsing buffer, *achieves near-perfect performance and consistently aligns instructions in excess of 90% of the time, over a wide range of issue rates. The performance boost provided by compiler optimization techniques is also investigated. Results show that compiler optimization can significantly enhance performance across all schemes. The* collapsing buffer *supplemented by compiler techniques remains the best-performing mechanism. The paper closes with recommendations and suggestions for future.*

1 Introduction

The recent MIPS R10000, Sun UltraSPARC and AMD K5 superscalar processors issue four instructions per cycle, with higher issue rates expected [1],[2],[3]. These processor designs employ multiple functional units and aggressive hardware scheduling to extract parallelism in the instruction stream. Next generation superscalar processors will most likely employ multithreading to further enhance parallelism. These highly parallel execution cores must be fed by sufficient instruction bandwidth, requiring optimized fetch unit design.

Fetching of instructions is constrained by three major factors: instruction cache performance, taken or indirect branches in the fetch stream, and instruction alignment. The design of the instruction cache has received much attention [4],[5],[6]. This body of work includes compiler techniques to enhance instruction cache performance [4],[7],[8]. The combined effect of this work is to lessen the impact of instruction cache misses on fetch bandwidth. Branch prediction is the second factor that constrains fetching. Several recent studies address the accuracy of branch prediction [9],[10],[11]. But branch prediction alone is not sufficient to deliver high fetch bandwidth. Even when branches are predicted accurately, the fetch unit must extract multiple, non-sequential instructions from the instruction cache in one cycle. The layout of instructions in the cache often frustrates this task. For high instruction bandwidth at high issue rates, the fetch unit must realign instructions in the predicted order, then pass the instructions on to the decode and execution units. Thus the third constraint on instruction fetch is due to the alignment of instructions in cache blocks. This problem is just emerging as issue rates increase beyond two instructions per cycle. This paper develops several solutions to the alignment problem.

Several approaches to high-bandwidth instruction

ISCA '95, Santa Margherita Ligure Italy
© 1995 ACM 0-89791-698-0/95/0006...$3.50

fetch have been implemented for commercial processors. Most decouple the instruction fetch unit from the execution unit via queues, and allow the fetch unit to speculate beyond branches [1],[12]. This decoupling reduces the impact of more-complicated (and higher-latency) instruction fetch hardware. In addition to this, the six instruction per cycle IBM POWER2 architecture employs an instruction cache with eight, independently-addressable banks [13]. This fetch unit can align many instruction sequences, but is limited by the POWER2's static branch prediction mechanism, which is known to have lower performance than dynamic schemes. The recently-announced AMD superscalar 29K addresses this limitation by embedding prediction and branch target address information in the cache array to enable a taken branch to be resolved without penalty [3]. However, this scheme cannot handle short branches within a cache block (e.g., hammocks), or multiple branches in one fetch, both of which are encountered frequently for integer code.

This paper presents several schemes of increasing complexity that address the instruction alignment problem. Implementation details are discussed for all the schemes. All comparisons are based on simulated results of the IPC for three microarchitectures. The results show that the most-complex scheme, the *collapsing buffer*, efficiently handles short forward branches and many cases of multiple branches. It achieves performance near the theoretical upper bound for a highly-parallel, 12 instruction issue microarchitecture. The effects of compiler optimizations on the performance of the schemes is also studied. The profile-driven code reordering optimization is found to be highly successful, significantly enhancing the performance of all schemes. A second optimization, *nop* insertion for branch target alignment, produces mixed results, suggesting this optimization plays only a secondary effect. The data is used to suggest several approaches for instruction fetch design at high issue rates.

The remainder of this paper is organized into three sections. The following section presents the machine model, the experimental technique, and other related assumptions employed in this study. This is followed by a discussion of the lower and upper bounds for instruction alignment performance. These bounds are termed *sequential* and *perfect* alignment, respectively. The designs and performance of the proposed hardware schemes are then discussed. The effect of compiler optimization is analyzed to find a balance between hardware and software solutions. The paper closes with recommendations and suggestions for future work in this area.

2 Experimental setup

The results that follow are presented for all six SPECint92 benchmarks, three additional integer benchmarks (`mpeg_play`, `bison`, and `flex`), and six SPECfp92 benchmarks. The benchmarks were compiled using GCC with the compiler options "`-O -fschedule-insn`." The latter option invokes a dag-based local scheduler. Experiments with this option show that it marginally enhances parallelism. All experiments were run using HP 9000/735-class workstations. The instruction set used for pipeline simulation is a simplified version of GCC's intermediate code captured after PA-RISC-specific register allocation but before final code generation. Instructions are encoded using a fixed, 32-bit format.

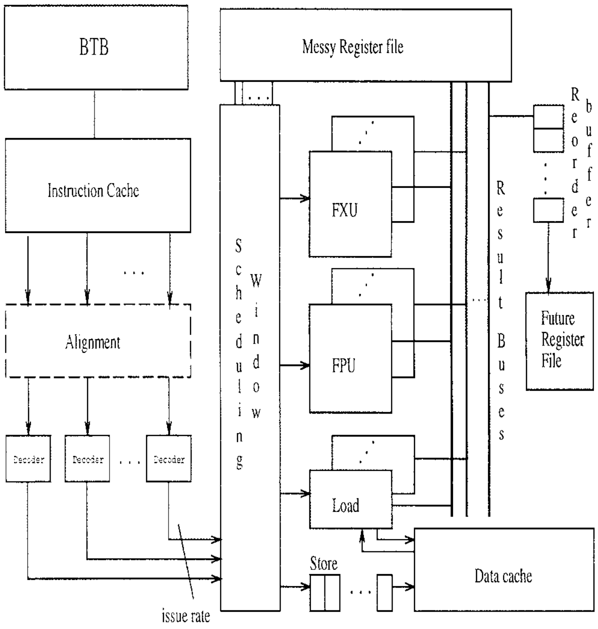

Figure 1: Structure of simulated microarchitecture.

Traces were captured using the *spike* tracing tool and then fed into a processor simulation. This simulation assumes a full-Tomasulo, out-of-order execution microarchitecture, depicted in Figure 1. Three versions of the microarchitecture are discussed in this paper, and their parameters are summarized in Table 1. All three versions have a scheduling window that resolves dependencies and implements Tomasulo-style renaming via tags. Entries in this window correspond to generic reservation stations. This window also serves to decouple the fetch unit from the execution unit, allowing the fetch unit to speculate ahead in the instruction stream. Speculative execution of more than one predicted conditional branch is supported via the precise interrupt facility (see below). The three classes of microarchitectures support differing degrees of speculation, in proportion to their issue rates. For example, the PI4 microarchitecture issues four instructions per cycle. Experiments with the degree of speculation showed that speculative execution beyond two branches was required to keep the pipeline full. Similarly, the PI8 architecture supports speculation beyond four, and the PI12 supports speculation beyond six branches.

Independent instructions are fired from the window into the execution core, which is composed of fixed-point units (FXUs), floating-point units (FPUs), branch units, and the data cache interface. Access to the data cache is through load units and a store buffer. Data cache misses are not explicitly modeled in the simulator. The PI4 model has two fixed-point units (FXU's), two floating-point units (FPU's), and two branch units. The PI8 model is similar, but scaled by doubling its resources to create a more parallel microarchitecture. The issue rate is increased to eight instructions per cycle. The PI12 model follows this design pattern, with an issue rate of 12 instructions per cycle.

Completing instructions are distributed via result buses. The number of result buses equals the total number of function units, so that bus contention seldom occurs. Two register files are maintained: the *Messy register file* and the *Future register file*. The former is used for out-of-order execution. If used without augmentation, the microarchitecture would be limited to imprecise interrupts. This is remedied using a reorder buffer [13]. The chief performance metric is *instructions retired per cycle* (IPC), which is the number of instructions leaving the reorder buffer (i.e., retiring) per simulated execution cycle.

All three microarchitectures have direct-mapped instruction caches. The cache block size is calculated so that a block holds the maximum issue rate of instructions. PI4 has size 16B, PI8 has size 32B, and PI12 has size 64B blocks. The cache sizes are also scaled with issue rate: 32KB (PI4), 64KB (PI8), and 128KB (PI12).

A branch-target buffer employing a 2-bit counter predictor is used for this study. The buffer is direct-mapped and has 1024 entries, comparable to commercial BTB designs (e.g., 512 entries for the Pentium [14], or 256/512 entries for the decoupled PowerPC 604 BTB [15]). Branch target addresses are also cached in the BTB for each entry. The BTB is interleaved into multiple banks with an interleave factor equal to the number of instructions in a cache block (e.g., an interleave factor of 4 for PI4). BTB interleaving is discussed further below.

Table 1: Machine model parameters: PI4, PI8, and PI12.

PI4 Machine model	
Issue rate	**4** instructions/cycle
Window queue	**16** entries
Instruction cache	32KB, dir. mapped, 16B blocks
Fixed-point unit	**2**, with latency = **1** cycle
Floating-point unit	**2**, with latency = **2** cycles
Branch unit	**2**, with latency = **1** cycle
Speculation	Speculates beyond **2** branches
PI8 Machine model	
Issue rate	**8** instructions/cycle
Window queue	**24** entries
Instruction cache	64KB, dir. mapped, 32B blocks
Fixed-point unit	**4**, with latency = **1** cycle
Floating-point unit	**4**, with latency = **2** cycles
Branch unit	**4**, with latency = **1** cycle
Speculation	Speculates beyond **4** branches
PI12 Machine model	
Issue rate	**12** instructions/cycle
Window queue	**32** entries
Instruction cache	128KB, dir. mapped, 64B blocks
Fixed-point unit	**6**, with latency = **1** cycle
Floating-point unit	**6**, with latency = **2** cycles
Branch unit	**6**, with latency = **1** cycle
Speculation	Speculates beyond **6** branches
Parameters common to all machine models	
Interlocking	Full Tomasulo, out-of-order
Branch target buffer	1024-entry buffer, 2-bit counter

3 Hardware Fetch Mechanisms

The lower bound for instruction fetch bandwidth is one instruction per cycle in the presence of a cache hit and a correctly predicted branch. However, few fetch mechanisms perform so poorly. A more-realistic lower bound is the performance of a sequential block fetch scheme. Such a scheme fetches an entire cache block and then selects multiple instructions from the block. This removes the normal cache word select logic and replaces it with masking logic. No hardware is provided to handle short branches inside the block (*intra-block branches*). The only code sequences that are handled by the technique are sequential instructions. For this reason, the technique will be called *sequential* throughout this paper. The operation of *sequential* is depicted in Figure 2 for a short program fragment.

The upper bound of instruction fetch bandwidth is when the pipeline is never starved due to a lack

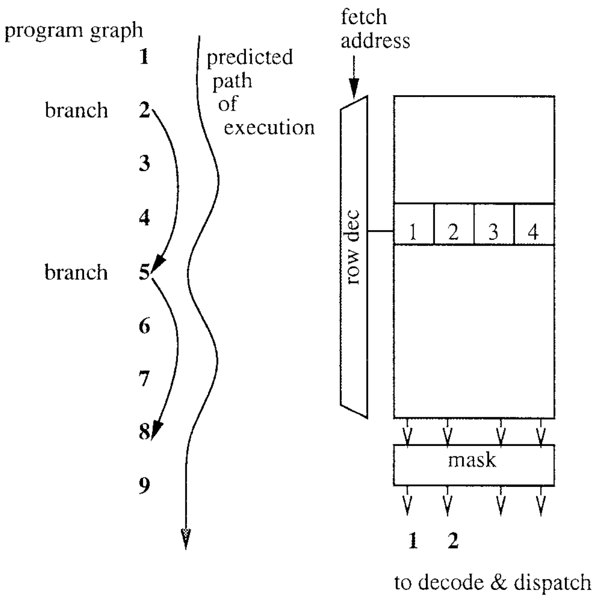

Figure 2: Example operation of *sequential* for sequence **1**, **2**, **5**, **8**.

of instructions. This bound is referred to as *perfect*. Specifically, *perfect* assumes that the instruction memory bandwidth into the scheduling window is unlimited (in the absence of instruction cache misses). Figure 3 presents the harmonic mean of the IPC for *sequential* and *perfect* for the integer and floating-point benchmarks. The data justifies the need for better instruction fetching for all machines, with the possible exception of floating-point code executing on the PI4 architecture. The loop-intensive floating-point benchmarks exhibit regular access patterns, reducing the need for better fetch mechanisms. The integer benchmarks require more effective mechanisms for better performance, due to a higher dynamic frequency of branch instructions.

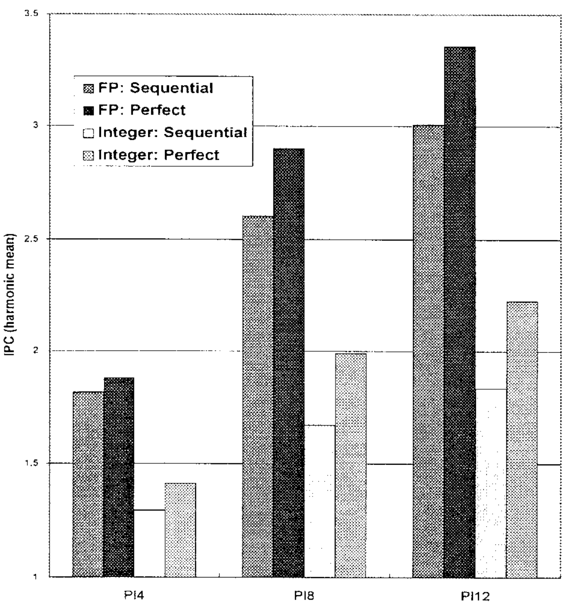

Figure 3: Performance of *sequential* versus *perfect* for integer and floating-point benchmarks.

3.1 Interleaved sequential

One enhancement to *sequential* is to interleave the instruction cache into two banks and prefetch one sequential block in advance. This *interleaved sequential* scheme (Figure 4) achieves higher effective issue rates over plain *sequential* for accesses that span block boundaries. Non-sequential accesses are not allowed. For example, if the sequence were **1**, **2**, **5**, **8**, as in Figure 2, the hardware would not be able to remove the useless instructions between **2** and **5**. As another example, assume *interleaved sequential* is fetching instructions in cache blocks A, B, C, D, etc. The cache is accessed for A and B, then B and C, then C and D, etc.

The *interleaved sequential* scheme must determine and eliminate any predicted non-sequential instructions before forwarding to the decoder. This is accomplished using a BTB interleaved by the number of instructions in a cache block [9]. A BTB query returns the successor block address and a bit-pattern predicting which instructions in the fetched block are valid for decoding. The successor block address is used to invalidate the sequential prefetch block. The block address and bit-pattern are found using a chain of comparators (depicted in Figure 5). Delay through the chain is proportional to the number of instructions in a cache block times the comparator propagation delay. (If this is significant, the chain can be redesigned using generate/propagate logic to reduce the delay.)

Two additional hardware entities are included to assist instruction aligning. These are the *interchange switch* and the *valid select* logic. The *interchange switch* can reverse the order of the fetch block and the target block. For example, if the fetch block is in the right-hand bank in Figure 4 and the target

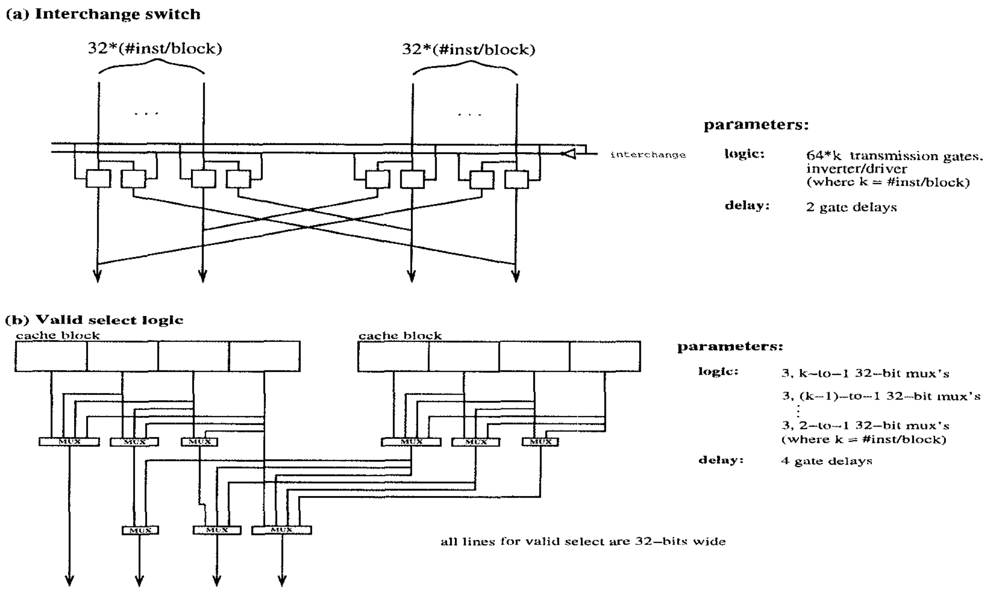

Figure 6: Design details of (a) the interchange switch, and (b) the valid select logic for *interleaved sequential* and *banked sequential.*

block is in the left-hand bank, the two blocks must be reordered so that instructions fed to the decoder are sequential. The design of the interchange switch that performs this task is shown in Figure 6(a). This design requires 64 × k transmission gates for cache blocks that hold k, 32-bit instructions per block.

The *valid select* logic has the responsibility of selecting the valid instructions from the two cache blocks. For an input of $2 \times k$ instructions, this logic selects the first k sequential, valid instructions as determined by the BTB prediction information. It requires an array of 32-bit multiplexers, and has nominal delay. The design of *valid select* is shown in Figure 6(b) (the right-most multiplexer is only required for *banked sequential*, which is described below).

Interleaved sequential is pipelined into three stages: *BTB, Cache*, and *Interchange-Valid.* There is bypass logic between the *BTB* and *Cache* stages so that the fetch pipeline latency for a mispredicted branch is two cycles, rather than three[1]. Since the typical length of instruction runs between branches is approximately four to six instructions, *interleaved sequential* does not perform well for high issue rates. This scheme can be enhanced by hardware that allows fetching to proceed across a branch.

3.2 Banked sequential

The *banked sequential* scheme is a modification of *interleaved sequential* to allow a limited amount of across-branch fetching. The hardware configuration is very similar to the former scheme (Figure 4). Alignment is possible only when the branch and its destination reside in different memory banks (inter-block branches). The hardware cannot handle intra-block branches. For a given fetch address, *banked sequential* finds the *likely successor address* then accesses the cache simultaneously for both the fetch block and its successor block. The likely successor address is determined by querying an interleaved BTB, as was done with the *interleaved sequential* scheme. Bank interference can occur if the successor block is in the same bank as the fetch block. In such a case, the successor block is not fetched.

Pipelining of banked sequential is similar to interleaved sequential, where the interchange switch and valid select form one stage of the three-stage pipeline. The BTB does not need to be queried again for the successor (prefetch) block. This is because as the fetch and successor blocks are being looked up in the cache (the second pipeline stage), the next instruction fetch queries the BTB with the successor

[1] The total misprediction penalty is the sum of the fetch misprediction penalty plus the number of cycles between when the branch is decoded and when it retires from the reorder buffer. This second component is instruction stream dependent and is modeled by the simulator.

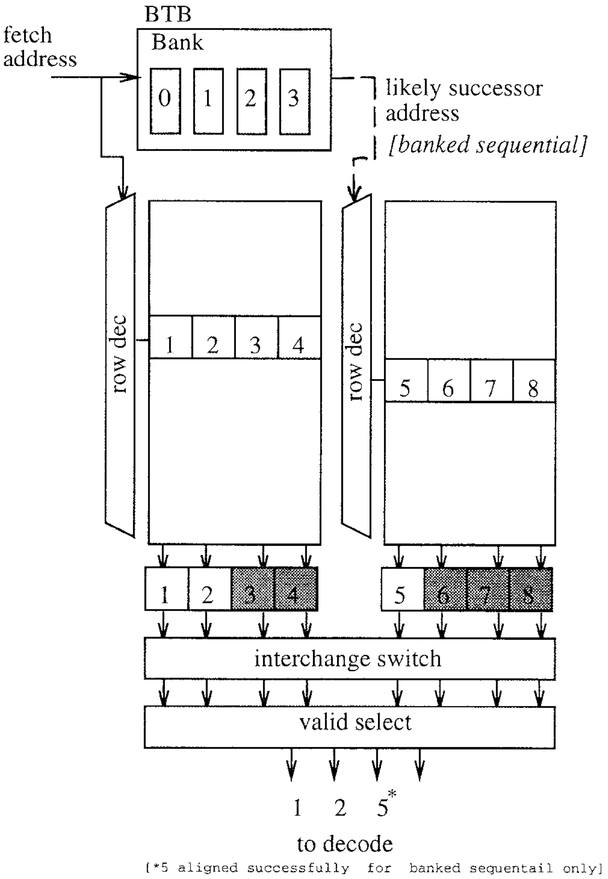

Figure 4: The *interleaved sequential* and *banked sequential* schemes.

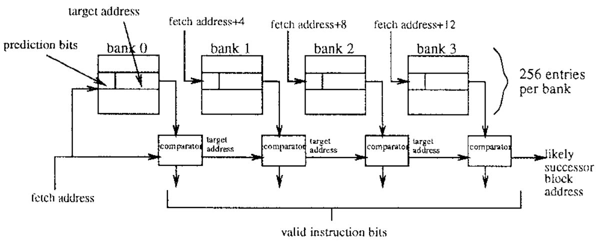

Figure 5: The interleaved BTB design (shown for PI4).

block address (the first stage). The BTB determines the successor block's valid bits with this overlapped cache/BTB access. Hence, the valid bits for the successor block are ready for use by *valid select* without two BTB queries.

Performance for *banked sequential* is limited by its inability to fetch across intra-block branches. The percentage of such branches to all taken branches for the workloads under study are shown in Table 2. For the PI4 machine (16-byte blocks), this percentage is small across all benchmarks except *compress* (14.58%). It increases dramatically as the block size increases. *Eqntott* increases from 6.13% to 29.26% from PI4 to PI8 (32-byte blocks). For PI12, almost half of the taken branches for *eqntott* (41.40%), *espresso* (45.68%) and *wave5* (41.73%) have their targets in the same block as the branch. This suggests the need for a mechanism to handle intra-block branches at high issue rates.

Table 2: Percentage of taken branches with target in the same block (*intra-block* branches).

Class	Benchmark	PI4	PI8	PI12
Int.	bison	6.05%	24.13%	30.81%
	compress	14.58%	14.59%	34.63%
	eqntott	6.13%	29.26%	41.40%
	espresso	1.40%	14.86%	45.68%
	flex	1.29%	3.88%	24.79%
	gcc	4.98%	14.08%	24.73%
	li	0.00%	5.74%	19.07%
	mpeg_play	0.70%	7.66%	11.96%
	sc	0.17%	11.02%	21.59%
FP	doduc	7.26%	11.85%	16.15%
	mdljdp2	0.26%	24.37%	66.10%
	nasa7	0.03%	0.06%	0.08%
	ora	0.01%	19.01%	23.16%
	tomcatv	0.08%	0.17%	13.97%
	wave5	2.71%	35.21%	41.73%

3.3 Collapsing buffer

The *collapsing buffer* scheme removes the useless instructions between an intra-block branch and it's target. It is an implementation designed to achieve *merging* [16], so that the target instruction follows the branch instruction in the decoder. This results in better decoder utilization and may also result in higher IPC.

The *collapsing buffer* scheme is shown in Figure 7. The BTB and cache are accessed in the same fashion as the previous two schemes. An additional buffer is

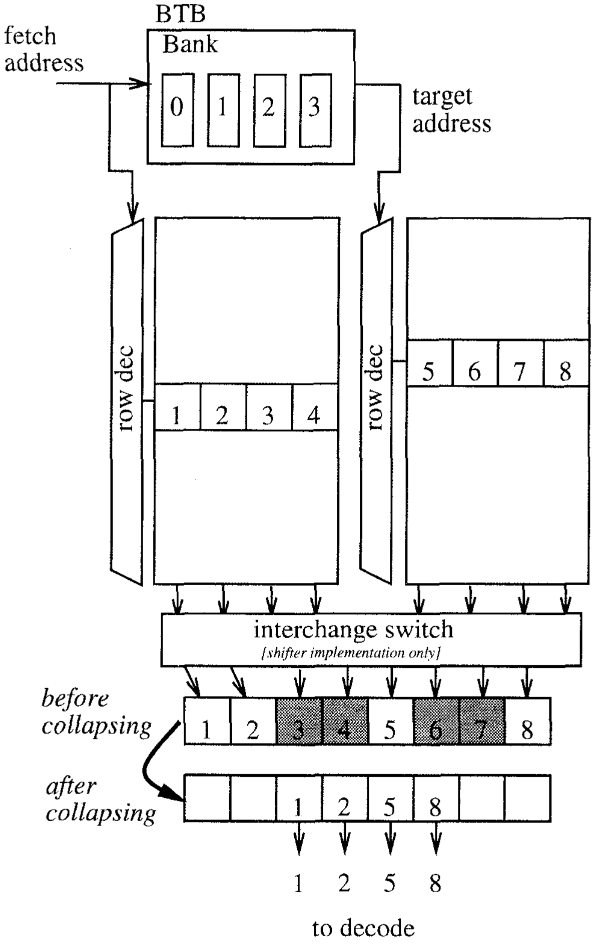

Figure 7: The *collapsing buffer* scheme.

added that collapses the gaps between valid instructions caused by intra-block branches. (Because of the capabilities of this buffer, the *valid select* logic of the previous two schemes has been removed.) Figure 8 details two possible implementations for the collapsing buffer. The first is a shifter-based implementation, and the second is a bus-based crossbar. The two implementations are open to tradeoffs based on area, speed and interconnect density. The crossbar implementation has the added advantage of being capable of handling backward branches, although this behavior was not supported by the controller modeled here.

Collapsing buffer is pipelined in a fashion similar to *banked sequential.* The crossbar implementation of the buffer removes the need for the *interchange switch* in addition to *valid select* logic. If traversal of the crossbar takes one cycle, the fetch misprediction penalty is two cycles. The shifter implementation will have a much higher misprediction penalty. Experiments with a penalty of three or more cycles produced little performance advantage for *collapsing buffer* over *banked sequential,* arguing against the shifter implementation (this is demonstrated below).

3.4 Performance of hardware schemes

The simulation results for *sequential, interleaved sequential, banked sequential,* and the *collapsing buffer* are shown in Figure 9(a) (integer benchmarks) and Figure 9(b) (floating-point benchmarks). Interleaving *sequential* provides a slight performance increase for both classes of benchmarks. Added fetch capabilities of the *banked sequential* and the *collapsing buffer* schemes provide distinct performance improvements, especially for integer benchmarks at higher issue rates. The floating-point benchmarks have well-behaved branches. Consequently, the performance of all the schemes for these benchmarks is relatively close for the PI4 machine. The need for more-sophisticated fetch mechanisms for floating-point code is more evident for the PI8 and PI12 machines, whose higher issue rates place a greater strain on the fetch unit.

The *collapsing buffer* is the most successful alignment mechanism across all processor designs. Floating-point benchmarks achieve almost perfect performance using this technique. Integer performance is also high, with IPCs very close to *perfect.* The justification for this scheme is provided by the difference in performance when compared to *banked sequential* for the PI12 machine. Here the gap in performance between the *collapsing buffer* and the other schemes is readily apparent.

Effective issue rate (EIR) is the rate at which instructions are successfully supplied to the decoders. For *perfect*, EIR is less than the ideal due to cache misses. For *sequential, interleaved sequential, banked sequential,* and the *collapsing buffer*, EIR is less than EIR(*perfect*) due to alignment failures. The ratio EIR/EIR(*perfect*) captures the ability of each of the schemes to align data. This metric is presented for each of the four schemes in Figure 10(a) (integer) and Figure 10(b) (floating-point). The *collapsing buffer* is the most-consistent scheme for delivering high EIR compared with EIR(*perfect*). It retains high performance in spite of increased issue rates from PI4 to PI12. The other schemes decrease in relative efficiency with approximately the same behavior from PI4 to PI12. (This is true for both integer and floating-point benchmarks.) This demonstrates that

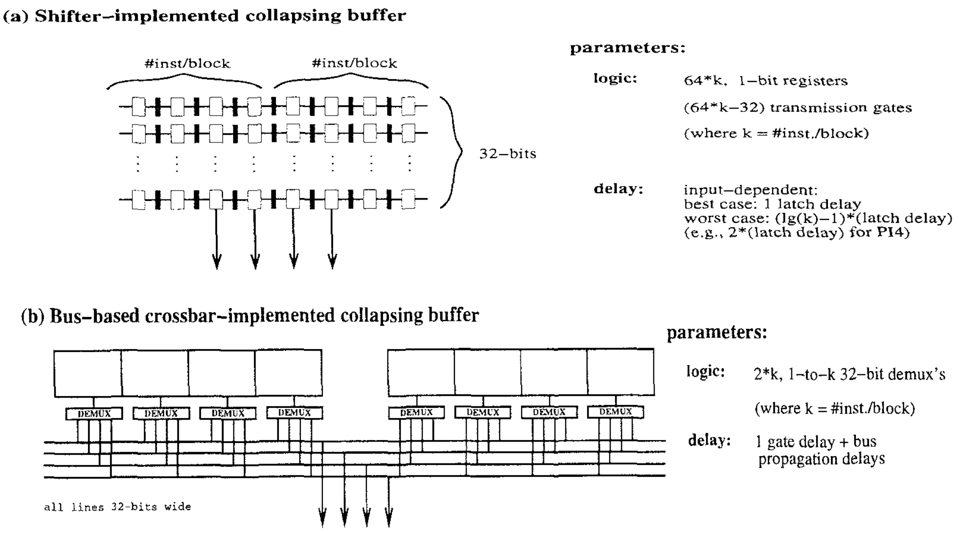

Figure 8: Design details of the collapsing buffer implemented (a) as a shifter, and (b) as a bus-based crossbar.

the *collapsing buffer* is a scalable alignment scheme, capable of delivering a high number of useful instructions in the presence of high issue rates.

In Section 3.3 it was mentioned that the shifter implementation of *collapsing buffer* does not provide much performance advantage over *banked sequential*. Figure 11 quantifies this observation. This figure is similar to Figure 9(a), except the *collapsing buffer* was simulated with a fetch misprediction penalty of three cycles. (This is perhaps the best-case performance for the shifter implementation.) *Banked sequential* actually performs slightly better than the *collapsing buffer/shifter* for PI4, and only slightly worse for PI12. This suggests that a low-misprediction-penalty implementation such as the crossbar is required to benefit from alignment using *collapsing buffer*[2].

4 The Effects of Compiler Optimizations

The hardware schemes presented above are limited by their ability to fetch across taken branches. Reduction of the number of non-sequential instruction accesses can lessen the impact of this limitation. The dynamic occurrence of taken branches can be reduced via compiler optimizations such as trace or superblock scheduling [17],[18]. These techniques reorder the code at compile time to form groupings of basic blocks that tend to execute sequentially. These larger groupings can be used to improve instruction cache performance, expand the scope of code scheduling, and enhance traditional optimizations [4],[7],[8],[19].

The effect of code reordering on the performance of the schemes was measured via simulation. Code reordering was performed on the benchmarks using trace selection and trace layout [7]. Six runs were performed for each integer benchmark. Each of the first five runs used a unique program input per run to generate profile statistics. These profiling inputs were taken from the input sets supplied by SPEC (or in the case of *li*, from student LISP assignments). An additional test input, not a member of the first five, was then used for the processor simulations. (SPECfp92 benchmarks were excluded since their code sequences are already highly-sequential in nature.)

The results of the simulations are presented in Figure 12. The figure also includes the performance of *sequential* and *perfect* without reordering (i.e., from the previous section), labeled as *sequential(unordered)* and *perfect(unordered)* in the figure. In general, code reordering significantly enhances performance. The success of code reordering can be attributed to a significant reduction in the number of taken branches. The percent reduction is shown in Table 3. The taken branches for a majority of the benchmarks are reduced by at least *20%*, and range from 15.72% for *li* to 44.2% for *compress*.

A detailed analysis of the data (Figure 12) reveals several interesting conclusions. *Sequential(reordered)*

[2] This observation is a function of the accuracy of the branch predictor.

(a)

(b)

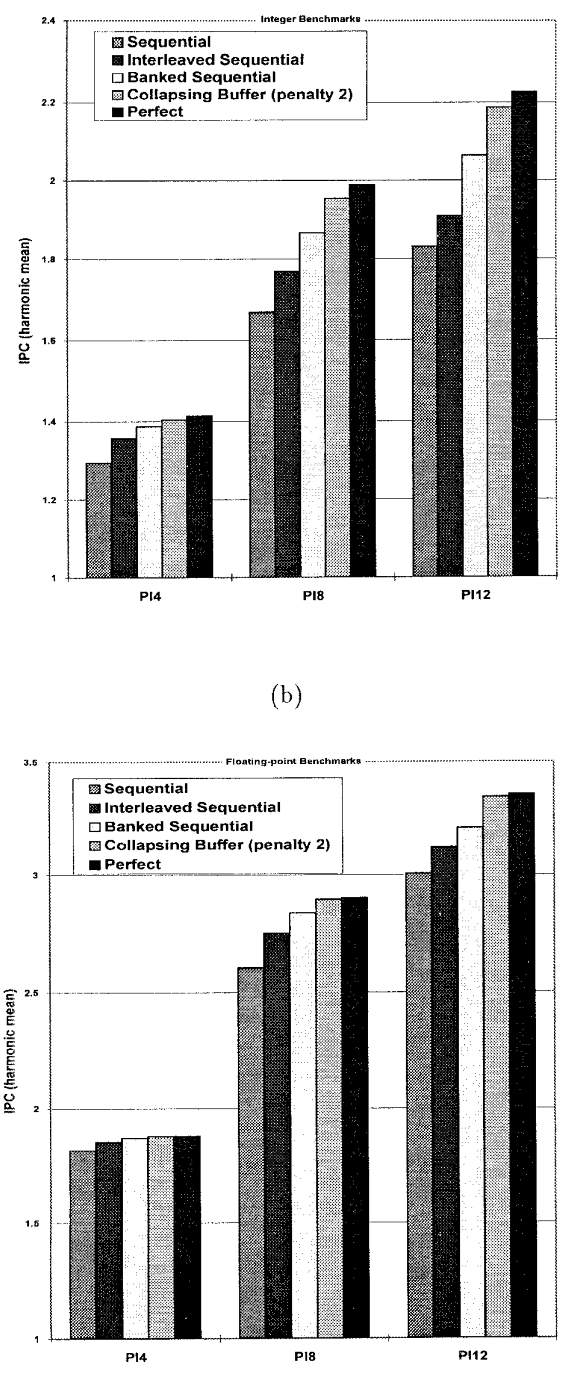

Figure 9: Performance of the alignment mechanisms for (a) integer, and (b) floating-point benchmarks.

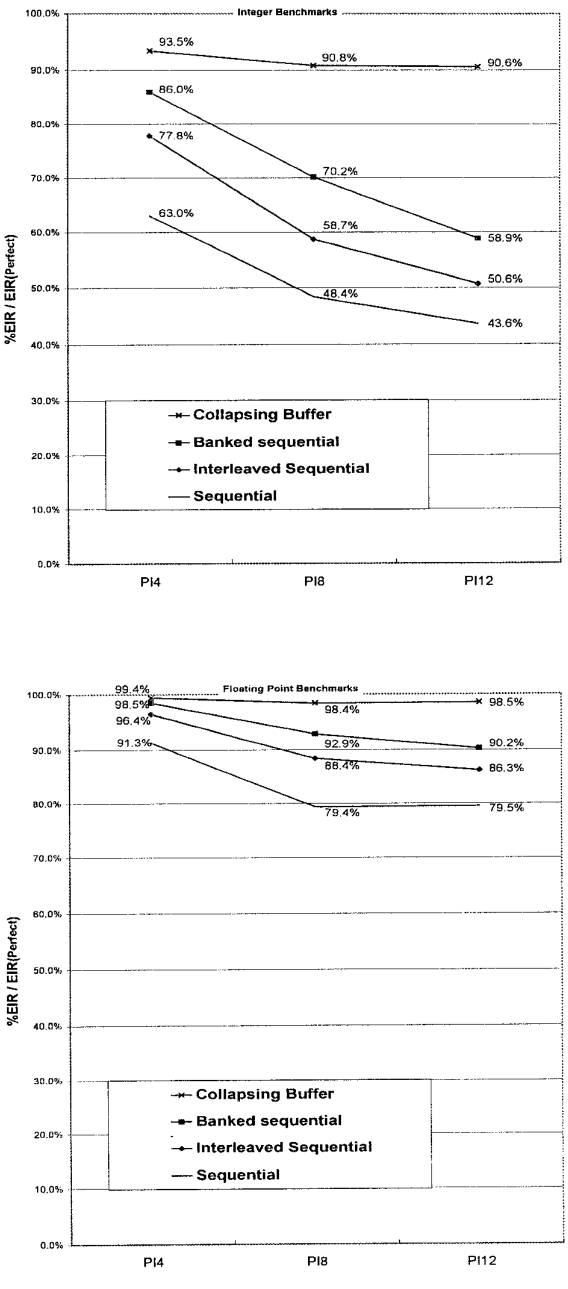

Figure 10: Percent EIR/EIR(*perfect*) of the alignment mechanisms for (a) integer, and (b) floating-point benchmarks.

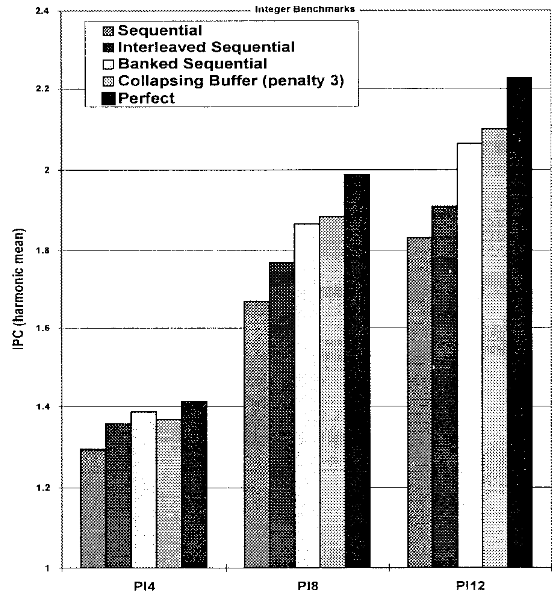

Figure 11: Performance of comparison of collapsing buffer assuming a three-cycle fetch misprediction penalty for integer benchmarks (all other schemes are shown with two-cycle penalties).

Table 3: Percent reduction in taken branches due to code reordering.

Benchmark	% Reduction
bison	25.26%
compress	44.20%
eqntott	24.52%
espresso	22.42%
flex	35.17%
gcc	37.20%
li	15.72%
mpeg_play	25.26%
sc	28.84%

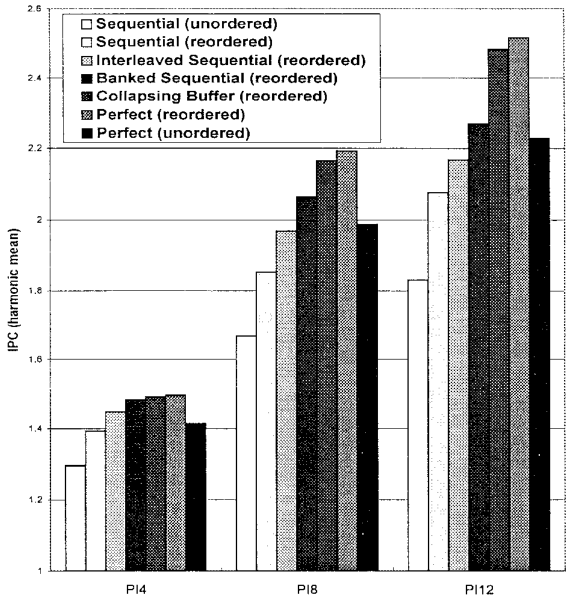

Figure 12: Performance of hardware schemes after code reordering.

achieves nearly the performance of *perfect(unordered)* for PI4. When reordered, the less-complicated *interleaved sequential* achieves comparable performance to *perfect(unordered)* across all three machine models. Hence, reordering can enhance the performance of *interleaved sequential* to match the performance of the hardware-only *collapsing buffer* scheme. However, when *collapsing buffer* is used with reordering, it nearly matches the performance of *perfect(reorderd)* from PI4 to PI12. This demonstrates that sophisticated compiler optimizations and sophisticated hardware combine to produce the highest performance for high issue rates.

4.1 Enhancing *sequential*

Reordering clearly enhances all hardware schemes. A compiler optimization to specifically enhance *sequential* is to align the traces by padding the end of each trace with *nop*s to force the following trace to begin at a cache block boundary [8],[20]. This scheme is termed *pad-trace*. *Pad-trace* can increase the number of useful instructions in each fetched block. In addition, Fisher's trace selection algorithm places likely-taken branches at the end of traces. Since these branches transition to the beginning of other traces, the inserted *nops* are seldom executed.

The disadvantage of both code reordering and *pad-trace* is that they require profile information, which is often hard to gather and requires additional steps when compiling code. (Hardware-based profiling techniques can remove many of these disadvantages, although their use was not studied in this paper. See [21].) An alternative to *pad-trace* is to pad all blocks without regard for trace membership. *Pad-trace* introduces significantly less *nops* than *pad-all*, as can be seen from Table 4.

Table 4: Degree of *nops* inserted for *pad-all* and *pad-trace* (expressed as percentage of *nops* vs. original code size).

block size 16B		
Benchmark	*pad-all*	*pad-trace*
bison	28.45%	2.22%
compress	29.53%	0.08%
eqntott	40.15%	7.17%
espresso	28.85%	5.60%
flex	27.75%	5.27%
gcc	32.31%	5.94%
li	33.20%	8.68%
mpeg_play	16.07%	3.45%
sc	37.89%	3.44%

block size 32B		
Benchmark	*pad-all*	*pad-trace*
bison	74.74%	5.35%
compress	74.98%	1.85%
eqntott	98.95%	16.77%
espresso	74.05%	12.93%
flex	67.65%	13.47%
gcc	80.33%	14.23%
li	80.33%	19.20%
mpeg_play	43.11%	8.87%
sc	90.71%	8.29%

block size 64B		
Benchmark	*pad-all*	*pad-trace*
bison	183.6%	12.28%
compress	190.8%	4.06%
eqntott	254.9%	41.37%
espresso	196.2%	30.50%
flex	173.6%	33.01%
gcc	214.0%	34.49%
li	225.1%	41.85%
mpeg_play	105.0%	21.18%
sc	237.8%	20.18%

The performance of *sequential* when augmented using *pad-all* and *pad-trace* is shown in Figure 13. Of the two, *pad-trace* achieves marginally higher performance improvement over its counterpart, *sequential(reordered)*, than *pad-all* achieves over *sequential(unordered)* for PI4. *Pad-all* achieves gains only for PI4, and experiences poor performance for processors using a larger cache size. This is due to the reduction in cache locality caused by excessive *nop* insertion. In general, *pad-all* appears to be unjustified even for PI4, since its benefit is more than offset by code expansion (Table 4). The code expansion for *pad-trace* is minor, justifying it as a refinement of code reordering.

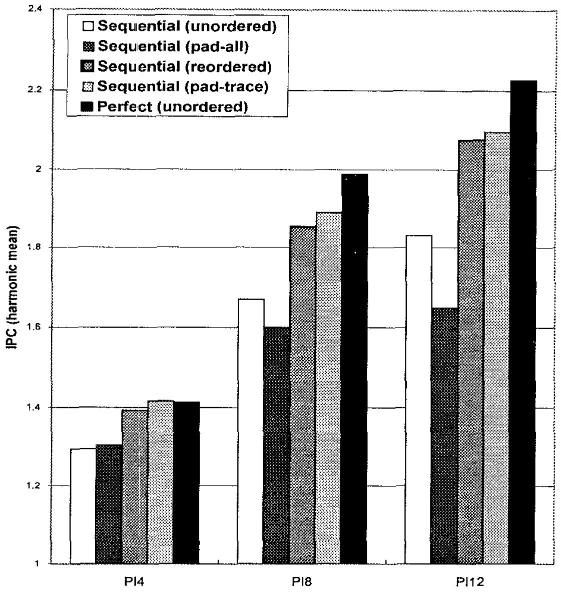

Figure 13: Performance of *pad-all* and *pad-trace* for *sequential*.

5 Concluding Remarks

The results presented in this paper demonstrate the need for efficient instruction alignment in order to support highly-parallel microarchitectures, such as PI8 and PI12. It appears that some fetch mechanisms such as *interleaved sequential* or *banked sequential* are also required for PI4 (which is similar in structure to several next-generation processors). The most robust scheme across all architectures studied was the *collapsing buffer*. The evidence for this is presented in the EIR/EIR(*perfect*) data of Figure 10. The *collapsing buffer* consistently aligns instructions at least 90% of the time.

The frequency of short branches within the same block motivated the design of the *collapsing buffer*.

Compiler-based techniques such as trace layout (reordering) can reduce this phenomenon by eliminating many taken branches. The data shows that these techniques can significantly enhance all schemes. For example, code reordering can enhance the performance of *interleaved sequential* to nearly match that of a hardware-only *collapsing buffer* approach for PI12. This also suggests that these techniques are applicable to existing machines. Padding with *nop*s, either used with reordering or used separately, produced only marginal improvements for *sequential* (the remainder of the hardware schemes were not significantly enhanced by padding). The best overall solution is to combine the highest-performance hardware scheme (*collapsing buffer*) with code reordering.

It remains to be seen what effect branch prediction accuracy has on the misprediction penalty when designing a pipelined *collapsing buffer*. Other, more sophisticated predictors do exist that have been designed for machines with high misprediction penalty [9]. Depending on the complexity of this branch prediction hardware, a shifter-based implementation of *collapsing buffer* may be viable.

Acknowledgements

We would like to thank Sumedh Sathaye, Ashutosh Singla, Prashant Maniar and the anonymous reviewers for their comments and suggestions on improving this paper. This research has been supported by AT&T.

References

[1] L. Gwennap, "MIPS R10000 uses decoupled architecture," *Microprocessor Report*, Oct. 1994.

[2] A. Agarwal, "UltraSPARC: A new era in SPARC performance," in *1994 Microprocessor Forum Proceedings*, Oct. 1994.

[3] M. Slater, "AMD's K5 designed to outrun Pentium," *Microprocessor Report*, Oct. 1994.

[4] S. McFarling, "Program optimization for instruction caches," in *Proc. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 183–191, Apr. 1989.

[5] D. B. Whalley, "Fast instruction cache performance evaluation using compile-time analysis," in *Proc. ACM SIGMETRICS '92 Conf. on Measurement and Modeling of Comput. Sys.*, (Newport, RI), pp. 13–22, June 1992.

[6] C. L. Mitchell and M. J. Flynn, "The effects of processor architecture on instruction memory traffic," *ACM Trans. Comput. Sys.*, vol. 8, pp. 230–250, Aug. 1990.

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.

[8] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[9] T. Yeh, *Two-level adaptive branch prediction and instruction fetch mechanisms for high performance superscalar processors.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 1993.

[10] B. Calder and D. Grunwald, "Fast & accurate instruction fetch and branch prediction," in *Proc. 21st Ann. International Symposium on Computer Architecture*, pp. 2–11, Apr. 1994.

[11] S. McFarling, "Combining branch predictors," WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[12] L. Gwennap, "PA-8000 combines complexity and speed," *Microprocessor Report*, Nov. 1994.

[13] S. Weiss and J. E. Smith, *POWER and PowerPC.* San Francisco, CA: Morgan Kaufmann, 1994.

[14] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[15] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," tech. rep., Somerset Design Center, Austin, TX, Apr. 1994.

[16] M. Johnson, *Superscalar microprocessor design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[17] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[19] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[20] M. Smotherman, S. Chawla, S. Cox, and B. Malloy, "Instruction scheduling for the Motorola 88110," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 257–262, Dec. 1993.

[21] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proc. 27th Ann. International Symposium on Microarchitecture*, (San Jose, CA), Nov. 1994.