Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques

Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, Douglas W. Clark

Departments of Computer Science and Electrical Engineering
Princeton University, Princeton, NJ 08540.
{skadron,psa,doug}@cs.princeton.edu, mrm@ee.princeton.edu

Abstract

Design parameters interact in complex ways in modern processors, especially because out-of-order issue and decoupling buffers allow latencies to be overlapped. Tradeoffs among instruction-window size, branch-prediction accuracy, and instruction- and data-cache size can change as these parameters move through different domains. For example, modeling unrealistic caches can under- or over-state the benefits of better prediction or a larger instruction window. Avoiding such pitfalls requires understanding how *all* these parameters interact.

Because such methodological mistakes are common, this paper provides a comprehensive set of SimpleScalar simulation results from SPECint95 programs, showing the interactions among these major structures. In addition to presenting this database of simulation results, major mechanisms driving the observed tradeoffs are described. The paper also considers appropriate simulation techniques when sampling full-length runs with the SPEC reference inputs.

In particular, the results show that branch mispredictions limit the benefits of larger instruction windows, that better branch prediction and better instruction cache behavior have synergistic effects, and that the benefits of larger instruction windows and larger data caches trade off and have overlapping effects. In addition, simulations of only 50 million instructions can yield representative results if these short windows are carefully selected.

Keywords

Microarchitecture, tradeoffs, branch prediction, cache, sampling, simulation, out-of-order execution, instruction window size, register-update unit

I. Introduction

Many microarchitecture studies focus on issues related to branch prediction, cache size, and more recently, instruction window size. Although points of diminishing returns exist, increasing the size of any of these structures generally increases performance. On the other hand, tradeoffs among these components are usually necessary due to die-size constraints.

Unfortunately, the relative importance and inter-relationships of these major parameters has become more complex as out-of-order issue and decoupling buffers cause latencies to interact in sometimes subtle ways. With a certain branch predictor, for example, adding data cache may be more beneficial than increasing instruction window size. More is better—but a small improvement to the branch predictor might be better yet, possibly shifting the tradeoffs so that adding instruction-window entries becomes more beneficial than adding cache. A branch-prediction study that models unrealistic caches can under- or over-state the benefits of better prediction. If the caches are too small, cache misses may mask the effects of branch mispredictions. Modeling caches that are perfect can be risky as well: some benchmarks stream data through even the largest caches, so omitting this significant contribution to execution time can lead to overstated benefits.

While such pitfalls may strike many readers as obvious, methodological mistakes like these are still common. Running pilot experiments to insure against such potential pitfalls and to cull the design space is expensive. Finally, detailed simulation generally prohibits executing realistic benchmarks to completion without reducing input sizes. We hope to address these problems by examining three issues in this paper.

- We characterize benchmark performance in terms of cache size and instruction-window size to help cull the design space that researchers need to explore. For example, several SPECint benchmarks [59] fit in 8 K instruction caches; while the small I-cache footprint of SPEC benchmarks is a common complaint, we provide detailed data showing this.

- We discuss tradeoffs among cache size, instruction-window size, and branch-prediction accuracy. When not taken into account, these tradeoffs can skew simulation results in damaging ways. For example, assuming perfect branch prediction makes large (*e.g.*, 128-entry) instruction-window sizes look artificially beneficial.
- Using time-series data that shows different phases of execution for cache miss rates and branch misprediction rates, we discuss the sensitivity of various tradeoffs to the choice of simulation window and of simulation length. This permits researchers to run shorter simulations and to avoid unrepresentative phases of execution. 50-million-instruction-long simulations are almost always adequate, provided the 50 M-instruction window is carefully chosen. In particular, short windows must not include the initial phases of the program.

To show these effects, this paper presents data for each SPECint benchmark that shows performance as a function of instruction-window size, data-cache size, and instruction-cache size, for both a contemporary hybrid branch predictor [36] and an ideal branch predictor. Comparing the ideal branch-prediction data against the hybrid branch-prediction data shows how much branch mispredictions limit performance; more importantly, it illustrates the circumstances under which simulating ideal prediction skews the results.

The paper also briefly considers how the tradeoff between data-cache size and instruction-window size changes as a function of branch prediction accuracy, briefly discuss return-address-stack repair, and presents some measurements for average branch-resolution times.

It is crucial to understand the interplay of these parameters, and this paper serves as a reference aid for this. We also hope that the comprehensive data we present will help researchers reduce their simulation requirements in various ways by describing a methodology for culling the simulation space and providing reference data for that purpose. We focus on the SPEC suite of benchmarks both because they provide consistency and comparability across studies, and because there are few other agreed-upon benchmarks that are portable and publicly available with source code. In particular, we focus only on the integer benchmarks because the SPEC floating-point programs have near-perfect branch prediction.

Section II describes our simulation approach and benchmarks in more detail. Section III examines the relationship between performance and instruction window size, and the impact of branch mispredictions on this relationship. Section IV examines the relationship between performance and both instruction- and data-cache size. Section V discusses branch prediction effects in more detail, and Section VI discusses simulation techniques. Section VII summarizes related work, and Section VIII concludes the paper.

II. Experimental Methodology

A. Simulator

We use *HydraScalar*—our heavily modified version of SimpleScalar's [4] *sim-outorder*—for our experiments. SimpleScalar provides a toolbox of simulation components—like a branch-predictor module, a cache module, and a statistics-gathering module—as well as several simulators built from these components. Each simulator interprets executables compiled by *gcc* version 2.6.3 for a virtual instruction set (the "PISA") that most closely resembles MIPS IV [43]. The simulators emulate the executable's execution in varying levels of detail.

Cycle-by-cycle simulators like HydraScalar that do their own instruction fetching and functional simulation (as opposed to relying on direct execution to provide instructions for simulation) can accurately simulate execution down mis-speculated paths. Like a real processor, HydraScalar checkpoints appropriate state as it encounters branches, and then upon detecting a mispredicted branch, wrong-path instructions are squashed, and the correct state recovered using the checkpointed state. Modeling the actual instruction flow on mis-speculated paths captures consequences like prefetching and cache pollution.

HydraScalar models in detail an out-of-order execution (OOE), 5-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, writeback, and commit. We add three further stages between decode and issue to simulate time spent renaming and enqueuing instructions. Issue selects the oldest ready instructions.

Table I summarizes our baseline model, which loosely resembles the configuration of an Alpha 21264 [26]. Our experiments vary instruction-window size, first-level data- and instruction-cache sizes, and branch-predictor accuracy. The simulations use a two-level, non-blocking cache hierarchy with miss-status holding registers (MSHRs) [29]. The cache module simulates a simple pipelined bus with fixed fetch spacing—the bus can accept a new transaction every n cycles, and once started the transaction completes without contention—and fixed memory latency (no interleaving, reordering, or fast-page-mode accesses). The model also assumes perfect write buffering (stores consume bandwidth but never cause stalls), which should have minimal impact on these results [51]. HydraScalar simulates a unified active list, issue queue, and rename register file. This type of instruction window is called a *register update unit* or

TABLE I

BASELINE CONFIGURATION SIMULATED BY HYDRASCALAR.

Parameter	Value	Comments
Processor core		
RUU (register update unit) size	64	
LSQ (load store queue)size	32	
Decode latency	3 cycles	Min time b/t fetch and issue
Fetch width	up to 4 instructions per cycle	Must be in same cache block
Decode width	up to 4 instructions per cycle	In-order
Issue width	up to 4 integer ops per cycle plus 2 FP ops per cycle	Out-of-order
Commit width	up to 4 instructions per cycle	In-order
Functional units	4 ALU/logical (1), 2 branch/shift(1), 1 integer multiply/divide (12/20), 1 FP add (4), 1 FP multiply (4), 1 FP divide/sqrt (16/33)	Latency appears in parentheses, and does not include the 1 cycle for reading registers
Memory ports	any combination of 2 memory ops	
Branch prediction		
Branch predictor	hybrid: 4 K $\times$ 12 global-history selector 4 K $\times$ 12-bit GAg predictor 1 K $\times$ 10-bit PAg predictor	uses 3-bit saturating counters
BTB	2048-entry, 2-way	updated only if taken
Return-address stack	32-entry	repaired after mis-speculation
Mispredict penalty	7 cycles	2 cycles for BTB-miss only
Memory hierarchy		
L1 data-cache	64 K, 2-way (LRU), 32 B blocks	8 MSHRs, 1-cycle latency
L1 instruction-cache	64 K, 2-way (LRU), 32 B blocks	2 MSHRs, 1 cycle latency
L2	unified, 8 M, 4-way (LRU), 32 B blocks,	4 MSHRs,12-cycle latency
Memory	100 cycle latency; 8 B/cycle	
L1→L2 bus	1 transaction every 2 cycles	
L2→mem bus	1 transaction every 8 cycles	
TLBs	128 entry, fully assoc., 30-cycle miss latency	both I-TLB and D-TLB

RUU [56]. The architectural register files (32 registers each for integer and floating-point) are separate and updated on commit. Using an RUU eliminates artifacts arising from interactions between active-list size and issue-queue size, and reduces the already large number of variables we examine. A load-store queue (LSQ) disambiguates memory references: stores may only pass preceding memory references whose addresses are known not to conflict. This study does not explore LSQ size, so for simplicity we fix the LSQ at one-half the RUU size to eliminate artifacts from stalls should the LSQ fill up.

HydraScalar uses a McFarling-style hybrid branch predictor [36] that combines two 2-level prediction mechanisms [65] with a selector that chooses between them. The two components are a 4K-entry GAg (global-history) predictor and a 1K-entry $\times$ 10 PAg (local-history) predictor; the latter uses 3-bit saturating counters. For each predictor, the selector chooses the component most likely to be correct by consulting its own 4K-entry table of saturating 2-bit counters, indexed by global history [7]. Since many PHT entries correspond to not-taken branches (or are simply idle), a BTB entry is only allocated for taken branches, permitting the BTB to have fewer entries than the PHT [5]. We have also added two varieties of perfect branch prediction. The first, which we call *100%-direction*, correctly predicts the direction of every conditional branch, but does not prevent any BTB misses. The second, *oracle prediction*, correctly predicts every branch and destination, regardless of type. The fetch stage we model makes a prediction for each branch fetched, but within a group of fetched instructions, those following the first predicted-taken branch are discarded because control must now jump to a new location. In practice, therefore, the fetch engine fetches through not-taken branches and stops at taken branches.

SimpleScalar updates the predictor state during the instruction-commit stage. This means there is a window of time,

TABLE II

BENCHMARK SUMMARY—WARMUP AND BRANCH STATISTICS.

	Warmup Insts	Branches per Instruction				Branch Accuracies			
		All	Return	Indir	Cond	All	Return	Indir	Cond
go	926 M	0.144	0.011	0.002	0.111	0.758	0.432	0.629	0.754
m88ksim	26 M	0.212	0.018	0.003	0.162	0.931	0.704	0.251	0.954
gcc (cc1)	221 M	0.194	0.015	0.030	0.144	0.827	0.546	0.350	0.861
compress	2576 M	0.202	0.028	0.000	0.133	0.925	0.993	0.063	0.888
li (xlisp)	271 M	0.236	0.027	0.082	0.137	0.906	0.721	0.814	0.918
ijpeg	824 M	0.059	0.001	0.003	0.051	0.894	0.728	0.984	0.879
perl	601 M	0.193	0.019	0.077	0.129	0.893	0.664	0.332	0.937
vortex	2451 M	0.166	0.021	0.021	0.121	0.968	0.901	0.768	0.980
tomcatv	2276 M	0.192	0.000	na	0.130	0.999	na	na	0.999

(Statistics are taken only from the post-warmup, 50 M-committed-instruction simulation window, and use the baseline configuration in Table I. "All" refers to all branches, whether conditional, direct-jump, indirect-jump, or return. "Indirect branches" here does not include returns. "Branch accuracy" refers to target-address prediction, except for the conditional-branch column, which presents direction-prediction accuracies.)

while a branch traverses the pipeline, during which its outcome is not available and the branch predictor uses slightly "stale" state. The prediction accuracies reported here are thus not as high as those reported in trace-driven prediction studies that do not model cycle-by-cycle timing effects. Although timing and implementation details of commercial microprocessors are difficult to come by, it seems that early—*i.e.*, speculative—history update is a recent innovation due to the fixup mechanisms required to undo corruption from incorrect updates. The importance of speculative update is discussed in [17], and mechanisms for implementing fixup appear in [24] and [52]. The Alpha 21264 implements speculative update with fixup for the global-history portion of its hybrid predictor [26].

Many mispredicted indirect jumps are function returns. Since a function might be called from many different locations, a BTB often provides the wrong target for these jumps. A return-address stack is a natural solution. It is best pushed and popped speculatively, in the fetch stage, and thus requires a fixup mechanism to prevent corruption. We model a simple mechanism, described later in Section V-B, that does not guarantee restoration of the correct state, but in practice virtually eliminates corruption [50]. All our simulations use this mechanism and, like the Alpha 21264, use a 32 entry stack [26].

Branch direction mispredictions suffer at least a seven-cycle latency, because the branch condition is not resolved until the writeback stage. The latency may be longer if the branch spends time waiting in the RUU. Conditional jumps for which the predicted direction is correct—and direct jumps—can still miss in the BTB (a *misfetch*), but a dedicated adder in the decode stage computes branch targets so that BTB misses can be detected early. A BTB miss therefore only experiences a 2-cycle penalty. Indirect jumps need to read the register file, which we assume cannot be done from decode. When the BTB mispredicts these targets, the error is only detected in the writeback stage.

B. Benchmarks

We examine the SPEC integer benchmarks [59], summarized in Tables II and III, and use the provided reference inputs. Like any other suite, SPEC has its shortcomings: for example, many of the programs fit in small I-caches. Because commercial workloads differ in some ways from the SPECint programs [35], users should exercise care in translating this paper's results to such programs. The trends and pitfalls identified here should hold true for many programs, but clearly not for all.

As mentioned in the introduction, we focus on the integer benchmarks because the floating-point programs have near-perfect branch-prediction accuracies, lower dynamic branch frequencies, and even smaller text sizes. We do include *tomcatv* as a reference for how floating-point programs might behave. All benchmarks are compiled using `gcc -O3 -funroll-loops` (-O3 includes inlining). Tomcatv is first translated to C using AT&T Bell Labs' *f2c* (1994 version). Simulations include all non-kernel behavior, like library code.

Some benchmarks come with multiple reference inputs, in which case we generally chose only one. For *go*, we chose a playing level of 50 and a 21x21 board with the `9stone21` input. For *m88ksim*, we used the `dhrystone` input; for *gcc*, `cccp.i`; for *ijpeg*, `vigo.ppm`; and for *perl*, we used the scrabble game. But for *xlisp*, we used all the

TABLE III

BENCHMARK SUMMARY—CACHE STATISTICS.

Benchmark	Mem Refs per Inst	Miss Ratios I$	D$	L2
go	0.299	0.004	0.005	0.000
m88ksim	0.301	0.000	0.000	0.000
gcc	0.453	0.008	0.010	0.000
compress	0.341	0.000	0.084	0.003
xlisp	0.405	0.000	0.016	0.000
ijpeg	0.271	0.000	0.008	0.000
perl	0.465	0.001	0.003	0.000
vortex	0.526	0.009	0.004	0.000
tomcatv	0.211	0.000	0.035	0.020

(L2 cache miss rates are global, that is, over all references made by the program.)

supplied LISP files as arguments.

We only perform full-detail simulation for a representative 50-million-instruction segment of the program, carefully selecting this window to capture behavior that is representative in terms of branch misprediction rate, cache miss rates, and overall IPC, and in particular, to avoid unrepresentative startup behavior. Section VI presents a detailed discussion of how we chose these simulation windows. Cycle-level simulations are run in a fast mode to reach the chosen simulation window. In this "warmup" phase, no pipeline simulation takes place; only caches, branch predictor, and architectural state are updated with each instruction. Then one million instructions are simulated in full-detail to prime other structures. Table II presents the total length of the warmup phase and summarizes branch behavior during the simulation window for each benchmark. Table III summarizes the behavior of the programs' memory references.

III. RUU SIZE AND BRANCH PREDICTION

Figure 1 presents, for each benchmark, a 3D graph showing instructions-per-cycle (IPC) as a function of both RUU size and L1 data-cache size. Each bar represents a single simulation with one combination of L1 size (plotted along the left-to-right horizontal axis) and RUU size (plotted along the front-to-back horizontal axis), and all other baseline parameters held constant. Our step sizes were chosen to yield a feasible number of simulations spanning a reasonable range of sizes. The IPC scale varies among benchmarks to best show the detail in each surface. Figure 2 presents similar data, but using the perfect direction predictor (BTB misses are still possible). It also extends the RUU axis to 256 entries. Note that the IPC scale changes between Figures 1 and 2.

We first consider performance as a function of RUU size, holding cache size constant and using the baseline branch predictor. The integer benchmarks fall into two groups, determined by the point at which performance as a function of RUU size plateaus. This plateau is the point at which additional entries support useful instructions so rarely that overall performance is barely affected. Most benchmarks show a plateau in performance at 48–64 RUU entries, regardless of cache size. These benchmarks all have branch prediction accuracy below 95%. *M88ksim* falls in this category as well; it achieves 95% accuracy and its performance has already plateaued at 32 entries. (At the end of this section we consider *m88ksim*'s decrease in IPC from 32 to 48 entries.) Note that contemporary processors also have RUU sizes in the 32–64 entry range. Thus, our results show that further increases in RUU sizes are unwarranted for many integer codes unless coupled with concomitant increases in branch-prediction accuracy.

TABLE IV

DIRECTION-MISPREDICTIONS PER INSTRUCTION.

go	0.0275	ijpeg	0.0060
m88ksim	0.0085	perl	0.0083
gcc	0.0202	vortex	0.0026
compress	0.0198	tomcatv	0.0000
xlisp	0.0117		

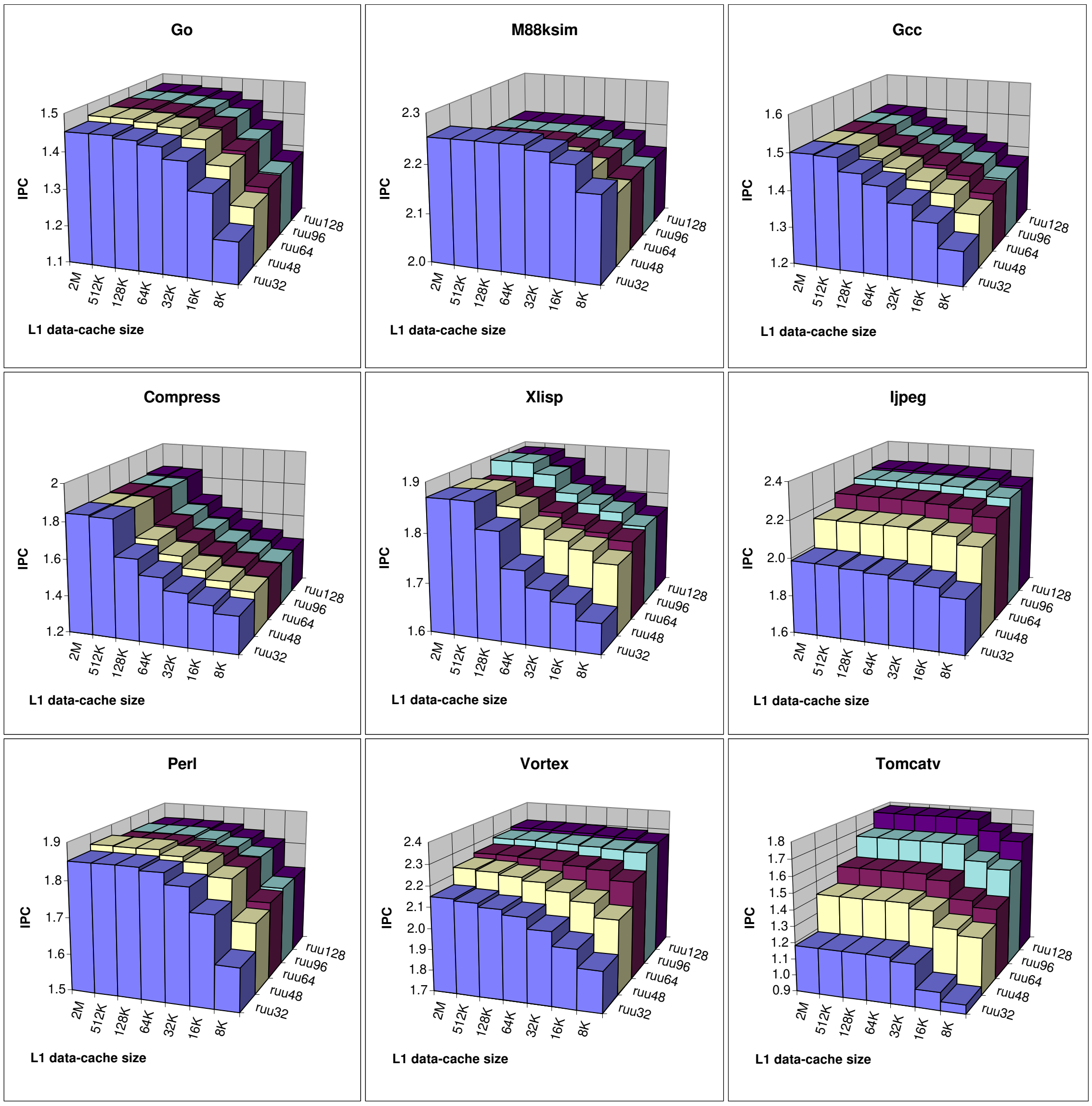

Fig. 1. IPC as a function of data cache size and RUU size with the baseline, hybrid branch predictor.

Two benchmarks, *vortex* and *ijpeg*, plateau later, at 96 entries. *Vortex* has a branch prediction accuracy of 98%, but *ijpeg*'s accuracy is only 88%. In fact, *ijpeg*, with one of the lowest accuracies, gets one of the largest gains among the SPECint programs from additional RUU. *Tomcatv*, with near-100% branch-prediction accuracy, gets by far the most benefit from additional RUU; going from 32 to 128 RUU entries improves performance by 50% with a 64 K cache.

These data suggest that conditional-branch prediction accuracy is a helpful indicator of sensitivity to RUU size: except for *ijpeg*, the poorly-predicted programs don't benefit from larger RUUs. A more accurate indicator also factors in basic block size to obtain *mispredictions per instruction* [15]. Table IV tabulates these rates. It shows that *ijpeg* benefits from further increments of RUU despite its low prediction rate because it has one of the lowest misprediction-per-instruction rates. Yet why do *m88ksim* and *perl* not benefit from deeper RUUs? They may have poor prediction accuracies, but they also have low values in Table IV. Poor prediction rates for *indirect* branches help explain the apparent discrepancies. Recall that, unlike a BTB miss for direct branches, which can be detected early, a BTB miss for indirect branches incurs a full misprediction penalty in our simulations. Table II shows that even though these benchmarks have among the lowest direction-misprediction-per-instruction rates, they have among the worst accuracies for indirect branches: 25% for *m88ksim*, and 33% for *perl*. Indirect-branch accuracies also help explain

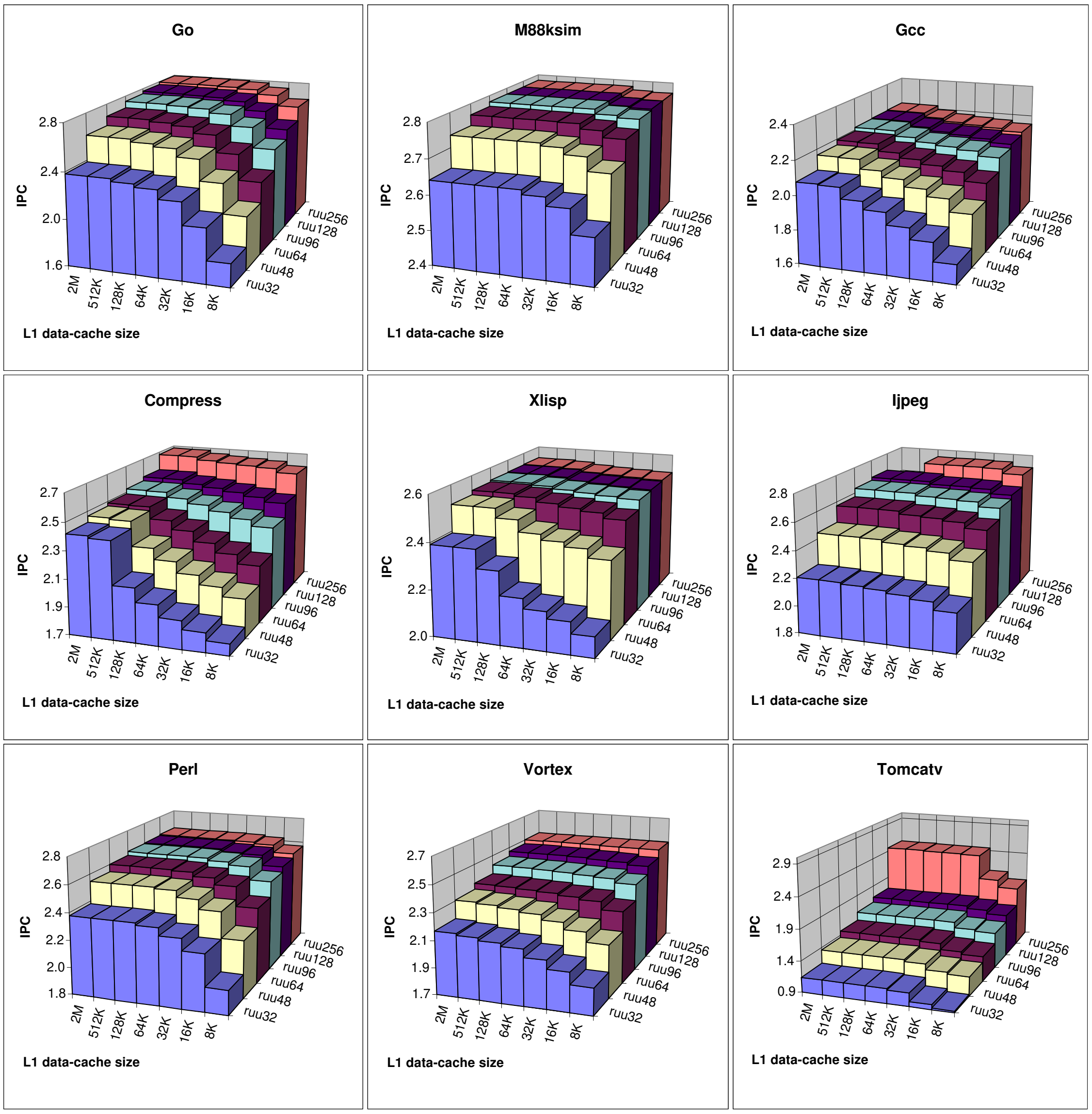

Fig. 2. IPC as a function of data cache size and RUU size with the 100%-direction-accuracy predictor. Simulator difficulties prevented us from obtaining the the two ruu256 values for ijpeg with large caches.

why *vortex* plateaus earlier than *ijpeg*: although both programs have low direction-misprediction-per-instruction rates, *vortex* only predicts 77% of indirect branches correctly, while *ijpeg* predicts 98% correctly.

The tradeoffs change as prediction accuracy improves and bigger RUUs are more likely to be filled with correctly speculated instructions. At the extreme, 100% direction accuracy (Figure 2) most programs derive significant benefit from bigger RUUs, even out to 256 entries.

A high misprediction rate limits the benefit of additional RUU entries because mispredictions prevent additional RUU entries from helping performance in two major ways. Deeper RUU entries often go unoccupied—mispredictions flush instructions so often that the deepest entries rarely even become active—and if occupied, deeper RUU entries often contain mis-speculated instructions. Note that early branch-predictor update (see Sec. II-A) would boost the prediction accuracy and moderately decrease the impact of these effects, while longer misprediction-resolution times (*e.g.*, longer pipelines) would increase the impact.

Figure 3 illustrates the first effect for *gcc*, which is fairly representative of the other benchmarks in this regard. The left-hand charts show, for a 128-entry RUU, the cycle-by-cycle distribution of RUU occupancies for our baseline branch predictor; the right-hand charts show the same data for 100%-direction prediction. An RUU entry is considered

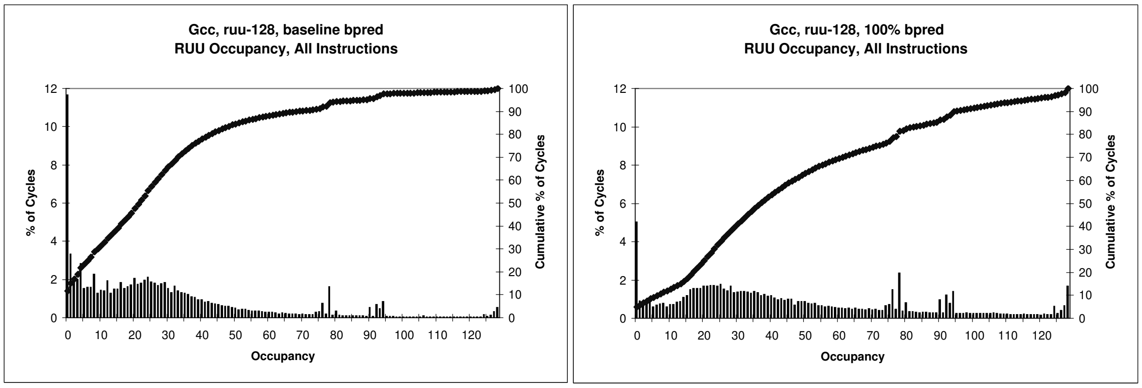

Fig. 3. Percentage of cycles in which the RUU contains a particular number of instructions. Data appears for both the baseline hybrid scheme and for perfect branch prediction. The per-occupancy bars use the left-hand y-axis; the cumulative curves use the right-hand y-axis. Any instruction in the RUU, even a mis-speculated one or one waiting to retire, counts toward the occupancy.

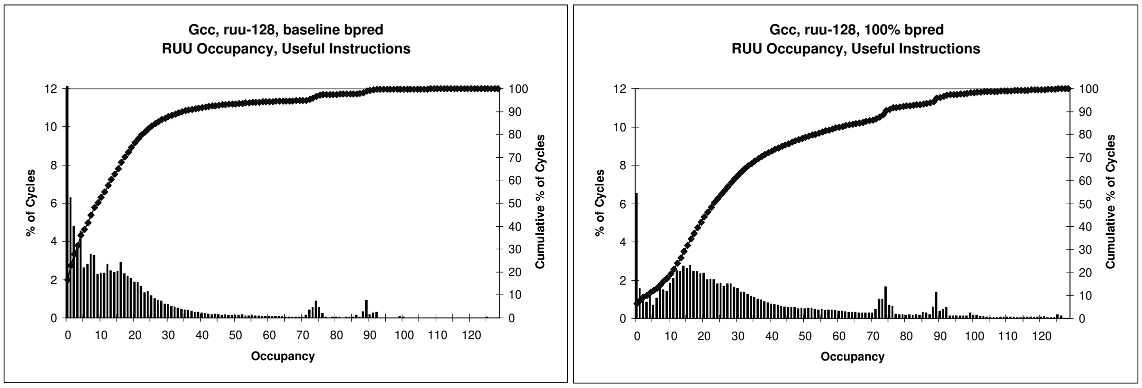

Fig. 4. Percentage of cycles in which the RUU contains a particular number of correctly-speculated instructions, for both the baseline and for perfect branch prediction. Only instructions that have not yet issued are counted. The y-axis here has been truncated: the 0 column has a value of 16.5%.

occupied in a cycle if it contains any instruction, whether already executed or not, and whether mis-speculated or not. The bars use the left y-axis, and show percentage of cycles that the RUU has a specific occupancy. The graphs show cumulative curves of the same data using the right y-axis.

For *gcc* with imperfect prediction and a 128-entry RUU, entries beyond the 64th are occupied only 11% of the time, and entries beyond the 96th only 2% of the time. Half of this large RUU is idle most of the time. In fact, although we do not show the data here, the deeper half of just a 48-entry RUU is idle more than 50% of the time. Deeper entries are used so infrequently that they can at best have a limited impact on performance. The low average occupancy results mainly from frequent squashing of instructions after a misprediction is identified. With perfect prediction, the picture looks quite different, as the right-hand side of Figure 3 shows. Even the deepest entries of a 128-entry RUU get regular use, and the RUU is often close to full.

With poor branch prediction, when a deeper RUU entry does become active, it often merely receives a mis-speculated instruction. These instructions can have helpful or harmful cache effects, but otherwise are useless and may cause contention for resources. Instructions which have completed execution and are waiting to retire also waste RUU entries. Figure 4 shows the cycle-by-cycle distribution of RUU occupancies for instructions that have been correctly speculated and have not yet executed. These correspond to issue-queue occupancy and are naturally lower than the overall occupancies in Figure 3. For imperfect prediction with *gcc*, useful occupancy exceeds 64 entries only 5.5% of the time, and exceeds 96 only 0.25% of the time. With 100%-direction prediction, on the other hand, useful occupancy for *gcc* exceeds 64 entries 15.5% of the time, and exceeds 96 entries 2.4% of the time. The change is more dramatic for other programs like *go*: with the baseline predictor, useful occupancy exceeds 64 entries not even 0.5% of the time, while with 100%-direction prediction, useful occupancy exceeds 64 entries 64% of the time.

Even if deep RUU entries receive a useful instruction, that instruction often waits for its operands. If the instruction waits so long that a smaller RUU could also have fetched it before the operands were ready, the earlier fetch and decode afforded by the larger RUU does not help. We find that instructions almost never issue from beyond the 48th–64th entry, but recall that deeper entries rarely contain any useful unissued instructions except with perfect branch prediction.

RUU size tends to matter more with smaller first-level data caches than with large caches, because smaller caches provide more miss latencies that can be overlapped with other instructions from the RUU. This is evident in Figure 1 for *go, gcc, compress, xlisp, perl, vortex*, and to a lesser extent, *ijpeg*: the slope along the RUU axis (front-to-back) becomes shallower as the data cache becomes larger. Some instances do arise, however, in which bigger caches cause RUU size to matter more. Consider the step from a 64-entry to a 96-entry RUU for *xlisp*: with an 8 K cache, Figure 1 shows no difference, while with a 512 K cache, a small improvement appears.

We studied two more consequences of branch mispredictions which might keep larger RUUs from helping. First, if a larger RUU causes branches to take longer to commit (because the branches are fetched earlier), branch-prediction accuracy may decline as a result. The branch predictor is updated when branches commit; between the time of a branch's fetch and its commit, subsequent branches will see slightly stale branch predictor state. Average branch-resolution time for *m88ksim* increases by 17% when moving from a 32 to a 48-entry RUU, and its conditional-branch accuracy in turn falls by 1%. We attribute *m88ksim's* better performance for a 32-entry RUU than for larger RUUs to this effect. The same effect may explain, in Figure 1, the small decline for *perl* for 64-entry and larger RUUs, and the small decline for *xlisp* for a 64-entry RUU with large caches.

Second, mis-speculated instructions can also have cache effects. These can hurt performance by creating contention or by causing extra instruction- or data-cache misses; on the other hand, and they can have a prefetching effect for caches [41]. The performance impact of these effects is less than 1% for all the benchmarks.

IV. Sensitivity to L1 Cache Size

A. Instruction Cache

Figures 5 and 6 show further 3D graphs of IPC as a function of cache size and RUU size; here, however, we vary I-cache size. Some of our benchmarks—*compress, ijpeg*, and *tomcatv*—fit even in an 8 K I-cache, and we omit plots for them. But for others, Figure 5 shows that instruction cache size profoundly affects performance. With the baseline branch predictor, increasing cache size from 8 K to 16 K improves performance by 20 to 40%—much more for *m88ksim*—and increasing cache size from 8 K to 128 K nearly doubles performance for *gcc, perl*, and *vortex*. So although the SPEC benchmarks are notorious for their small cache footprints, several still benefit from larger I-caches than those found in processors today.

I-cache size has even more impact with 100% direction-accuracy branch prediction, as seen in Figure 6. This is partly because perfect prediction eliminates the prefetching done by mis-speculated paths, but this mechanism only affects IPC by about 1%. Most of the difference is due simply to eliminating time wasted on mispredictions.

Tables V and VI present instruction-cache miss ratios for the baseline and 100% predictors respectively; RUU size in these tables is 64 entries. Note that cache miss ratios include wrong-path accesses, and recall that caches are 2-way set-associative. The higher miss rates in Table VI (100% prediction) result from the lack of pollution by wrong-path prefetching.

If enough instructions reside in the RUU, some I-cache misses can be partially or completely hidden with waiting instructions. But I-cache misses are strongly clustered: with an 8 K I-cache, from 45 to 70% of misses happen within 2 cycles of the previous miss, and with a 64 K cache, from 40 to 60%. (90% of *perl's* I-misses happen within 2 cycles, but it experiences only a few thousand misses during the simulated interval.) When misses are so strongly clustered, even a full RUU doesn't help much, so frequent I-cache misses introduce many bubbles. Data misses are less clustered, so this problem is not as prevalent for data references, and an RUU can do a better job of compensating for the cache misses.

The most important implication of these results is that small I-caches limit the value of a larger RUU. Figure 2 showed that with 100%-direction branch prediction, RUU size strongly affects performance. Those data assume a 64 K I-cache. But in Figure 6, shrinking the I-cache to 8 or 16 K largely wipes out RUU effects for *go, gcc, and perl*. The I-cache misses so often that it can rarely build a sufficient buffer of instructions. When it does, clustering of misses means that even a big and full RUU drains, and the processor still stalls.

This is an especially clear example of shifting tradeoffs: the choice of one design or simulation parameter can control the impact of another. For simulation purposes, this means certain parameters can have a profound impact on how realistic the results are.

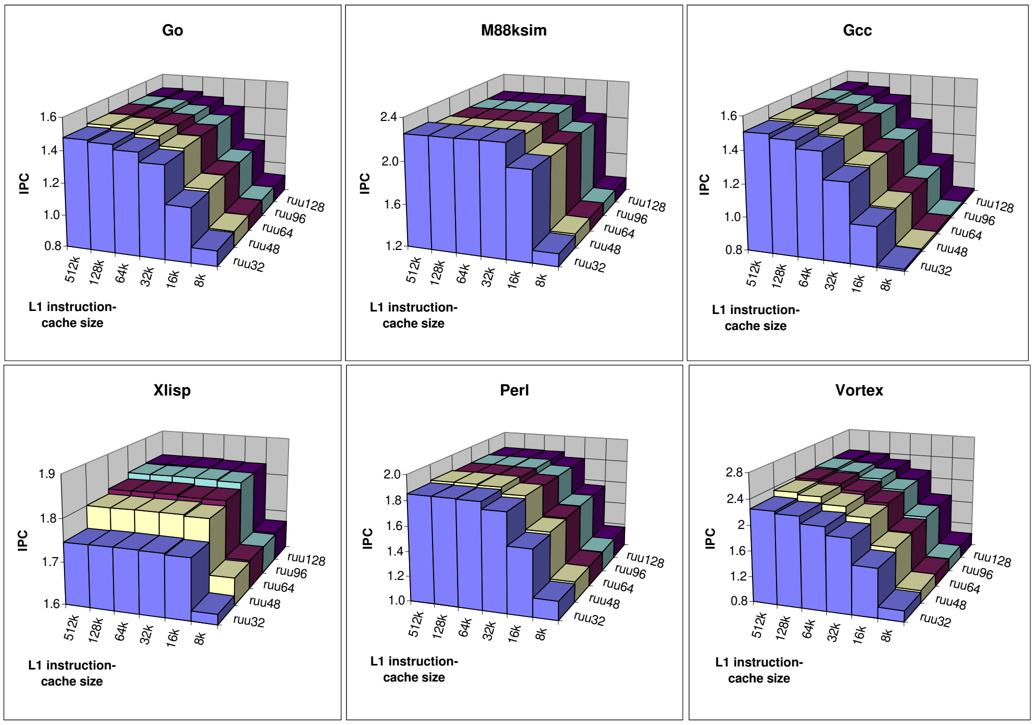

Fig. 5. IPC as a function of instruction cache size and RUU size with the baseline, hybrid branch predictor.

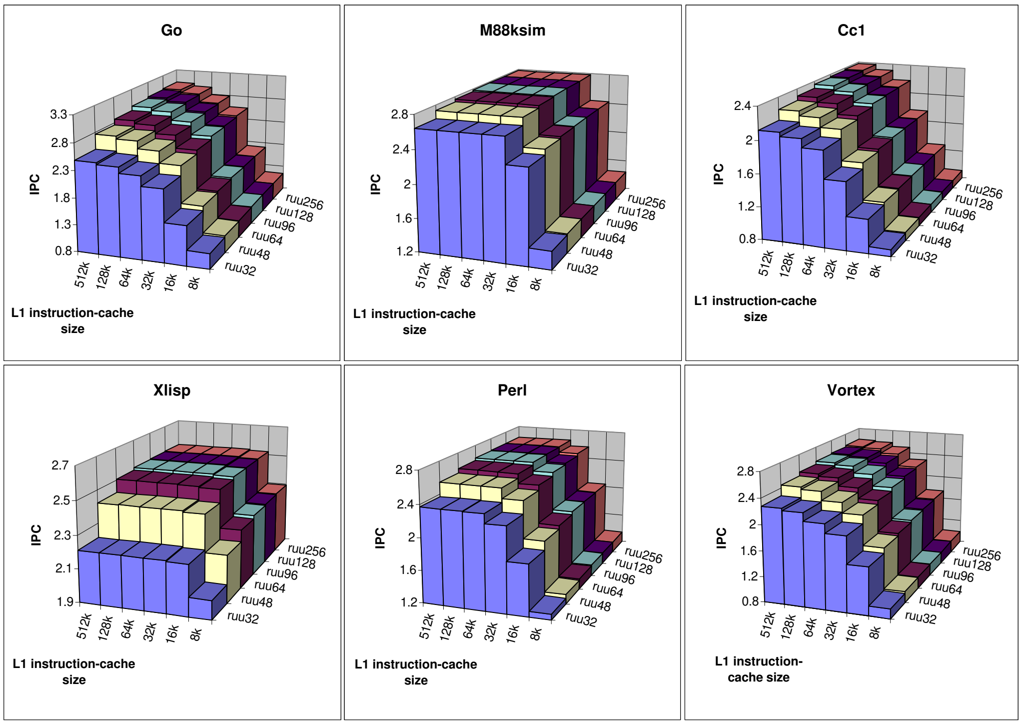

Fig. 6. IPC as a function of instruction cache size and RUU size with the 100%-direction predictor.

TABLE V

INSTRUCTION-CACHE MISS RATIOS, BASELINE HYBRID PREDICTOR, 96-ENTRY RUU.

	8 K	16 K	32 K	64 K	128 K	512 K	2 M
go	0.098	0.044	0.011	0.005	0.001	0.000	0.000
m88ksim	0.078	0.011	0.000	0.000	0.000	0.000	0.000
gcc	0.126	0.069	0.029	0.008	0.003	0.000	0.000
xlisp	0.009	0.000	0.000	0.000	0.000	0.000	0.000
perl	0.084	0.029	0.005	0.001	0.000	0.000	0.000
vortex	0.179	0.060	0.018	0.009	0.002	0.000	0.000

TABLE VI

INSTRUCTION-CACHE MISS RATIOS, 100% DIRECTION-ACCURACY PREDICTOR, 96-ENTRY RUU.

	8 K	16 K	32 K	64 K	128 K	512 K	2 M
go	0.143	0.068	0.017	0.007	0.002	0.000	0.000
m88ksim	0.077	0.012	0.000	0.000	0.000	0.000	0.000
gcc	0.146	0.079	0.032	0.008	0.003	0.000	0.000
xlisp	0.010	0.000	0.000	0.000	0.000	0.000	0.000
perl	0.095	0.032	0.006	0.000	0.000	0.000	0.000
vortex	0.162	0.056	0.019	0.009	0.002	0.000	0.000

I-cache size can influence how much effect RUU size has on IPC, but over the range we examine, RUU size does not influence how much effect I-cache size has on IPC.

B. Data Cache

In moving from a smaller RUU to a larger RUU, we might expect that the processor's sensitivity to cache performance would diminish, since there are more opportunities to overlap miss times with useful work. This presumes an RUU with enough correctly-speculated instructions. Turning back to Figure 1, with the baseline, hybrid branch predictor, this effect is mildly visible when moving from 32 to 48 RUU entries for *gcc, xlisp*, and more strongly over the entire graph for *vortex*. But "useful" RUU occupancy is generally quite low, so a larger RUU has minimal impact.

The effect is prominent, however, with 100%-direction prediction in Figure 2: every benchmark except *ijpeg* exhibits declining D-cache sensitivity as RUU size becomes deeper. For most benchmarks, D-cache hardly matters once the RUU is in 96–256 entries deep, depending on the benchmark.

Plotting performance as a function of cache size, as in Figures 1 and 2, also permits estimation of the programs' working set sizes. Table IX identifies the point at which further increases in cache size stop producing improvements in IPC—a good approximation of working set size. (Although some applications have a hierarchy of working sets [47].) Knowledge of working set sizes like this can be useful in three ways. It ensures that when choosing a single data cache size for some other type of study (*e.g.* a branch prediction study), the chosen operating point does not have an artificially high miss rate. Second, it helps establish a suitable range of data-cache sizes for simulation-based studies [63]. Third, knowledge of working set sizes helps when scaling cache or problem sizes.

Tables VII and VIII present data-cache miss ratios for the baseline and 100% predictors respectively. Recall that caches are 2-way set-associative. The tables verify that IPC closely correlates with L1 D-cache miss ratio.

C. Instruction Cache and Data Cache Interactions

We have examined interactions between data-cache size and RUU size, and instruction-cache size and RUU size. Figures 7 and 8 plot IPC as a function of data-cache and instruction-cache size together. The results follow naturally from our preceding results. Performance is strongly sensitive to I-cache size until the program fits or almost fits into the I-cache. Until the program fits in the I-cache, I-cache size matters substantially more than D-cache size. With 100%-direction prediction, data-cache size matters little because we have chosen such a large RUU. We have also run similar experiments with RUU size set to 32 entries, in which case performance is more sensitive to data-cache size, just as the "ruu32" data in Figure 2 suggests.

Just as too-small I-caches minimize the impact of changing RUU size, small I-caches limit the impact of larger D-caches. This trend is only visible in Figure 7; we use a 128-entry RUU and the D-cache curves are thus already flat in

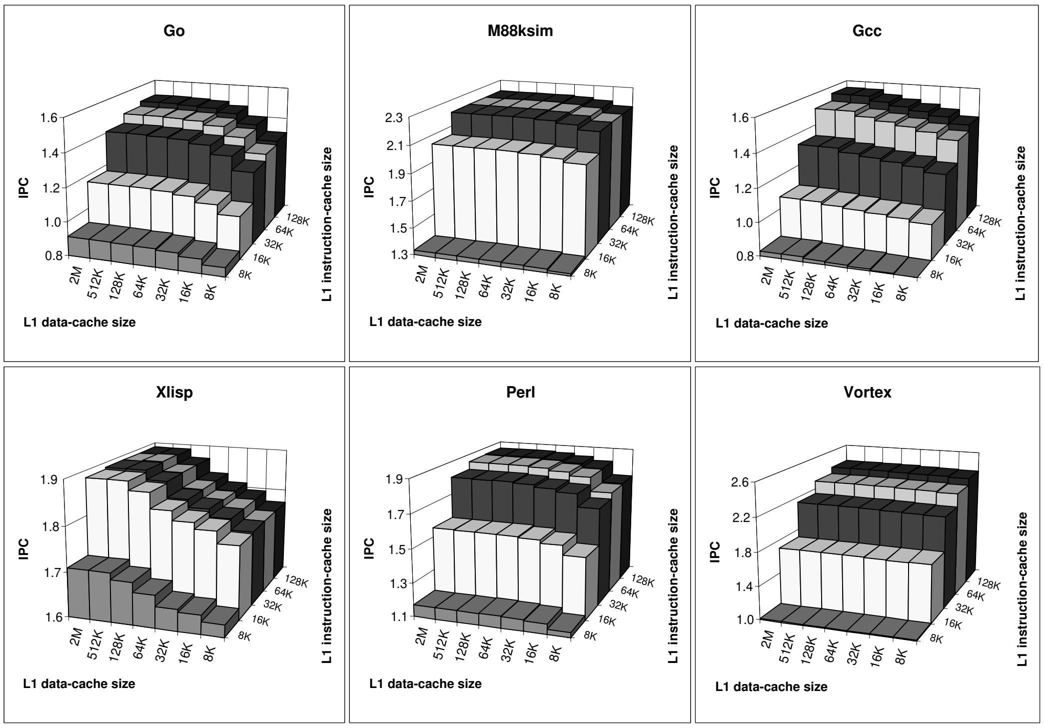

Fig. 7. IPC as a function of data-cache size and instruction-cache size with the baseline, hybrid branch predictor. RUU size here is 128 entries.

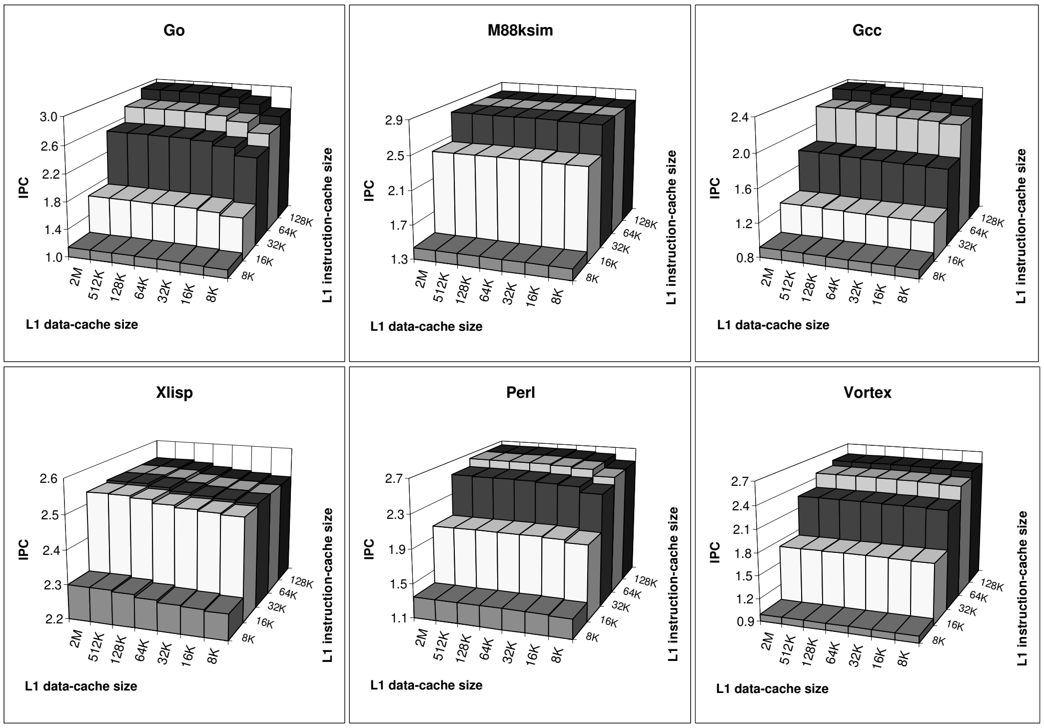

Fig. 8. IPC as a function of data-cache size and instruction-cache size with the 100%-direction predictor. RUU size is 128 entries.

TABLE VII

DATA-CACHE MISS RATIOS, BASELINE HYBRID PREDICTOR, 96-ENTRY RUU.

	8 K	16 K	32 K	64 K	128 K	256 K	512 K	1 M	2 M
go	0.073	0.034	0.014	0.005	0.002	0.000	0.000	0.000	0.000
m88ksim	0.007	0.003	0.001	0.000	0.000	0.000	0.000	0.000	0.000
gcc	0.041	0.027	0.017	0.010	0.005	0.002	0.001	0.000	0.000
compress	0.127	0.116	0.103	0.084	0.058	0.025	0.004	0.003	0.003
xlisp	0.028	0.023	0.021	0.016	0.007	0.000	0.000	0.000	0.000
ijpeg	0.049	0.035	0.011	0.008	0.007	0.006	0.005	0.005	0.005
perl	0.034	0.012	0.006	0.001	0.000	0.000	0.000	0.000	0.000
vortex	0.022	0.013	0.008	0.004	0.003	0.002	0.001	0.001	0.001
tomcatv	0.277	0.183	0.046	0.037	0.036	0.036	0.036	0.036	0.036

TABLE VIII

DATA-CACHE MISS RATIOS, 100% DIRECTION-ACCURACY PREDICTOR, 96-ENTRY RUU.

	8 K	16 K	32 K	64 K	128 K	256 K	512 K	1 M	2 M
go	0.084	0.037	0.015	0.005	0.002	0.000	0.000	0.000	0.000
m88ksim	0.008	0.003	0.001	0.000	0.000	0.000	0.000	0.000	0.000
gcc	0.045	0.029	0.019	0.010	0.006	0.002	0.001	0.000	0.000
compress	0.124	0.110	0.094	0.074	0.051	0.022	0.004	0.003	0.003
xlisp	0.030	0.025	0.022	0.018	0.007	0.000	0.000	0.000	0.000
ijpeg	0.050	0.036	0.012	0.008	0.007	0.006	0.005	0.005	0.005
perl	0.035	0.012	0.006	0.001	0.000	0.000	0.000	0.000	0.000
vortex	0.022	0.013	0.008	0.004	0.003	0.002	0.001	0.001	0.001
tomcatv	0.264	0.174	0.045	0.036	0.036	0.035	0.035	0.035	0.035

TABLE IX

DATA-CACHE WORKING SETS

go	64k	ijpeg	16k
m88ksim	64k	perl	64k
gcc	512k	vortex	128k
compress	512k	tomcatv	32k
xlisp	512k		

Figure 8. *Go, gcc*, and *perl* especially show how a too-small I-cache limits the benefit of larger D-caches. Many data and instruction misses presumably coincide, and the effect of eliminating some data misses is hidden if the instruction misses remain.

V. BRANCH PREDICTION AND THE RELATIVE IMPORTANCE OF RUU AND L1 DATA-CACHE

A. Direction-prediction accuracy

Sections III and IV compared the effects of RUU size and L1 data-cache size on performance, and considered both a realistic hardware branch predictor and a 100%-direction-accuracy predictor. The IPC vs. RUU size vs. data-cache size graphs in Figures 1 and 2 form the backbone of those observations. The change in moving from our baseline hybrid predictor to 100%-direction accuracy is substantial, and the corresponding IPC surfaces change dramatically between those two figures. This section therefore considers how the surfaces change as we vary branch prediction accuracy in finer steps.

Unfortunately, realistic hardware often cannot achieve accuracies near 100%: some branch behavior is simply too random. To see what might happen to RUU-cache tradeoffs in this range of accuracies, we boost the PHT's accuracy by randomly choosing some mispredictions and *artificially* correcting them. Conditional branch predictions modified in this way behave as though their directions had been correctly predicted. The simulator can be set to attain any desired direction-prediction accuracy by adjusting the fraction of repaired predictions. This artificial adjustment takes place infrequently, so most of the PHT's predictions are untouched. We do not claim here that a program would behave exactly as shown, since the mispredictions to fix up are chosen randomly. We hope only to suggest how the tradeoffs between RUU size and L1 data-cache size might change as branch-prediction techniques continue to improve. In fact, the tradeoffs change in a quite clear-cut fashion.

For the first boosting of prediction accuracy, we use a huge but realistic PHT instead of applying our artificial-boosting technique. This huge predictor is similar to the baseline predictor, but modified to XOR the history bits with some address bits, and with 64 K entries in each table. Subsequent steps in accuracy are achieved with synthetic boosting, but using the huge predictor minimizes the number of correct predictions that need to be artificially created. At 100% direction accuracy, only BTB misses remain.

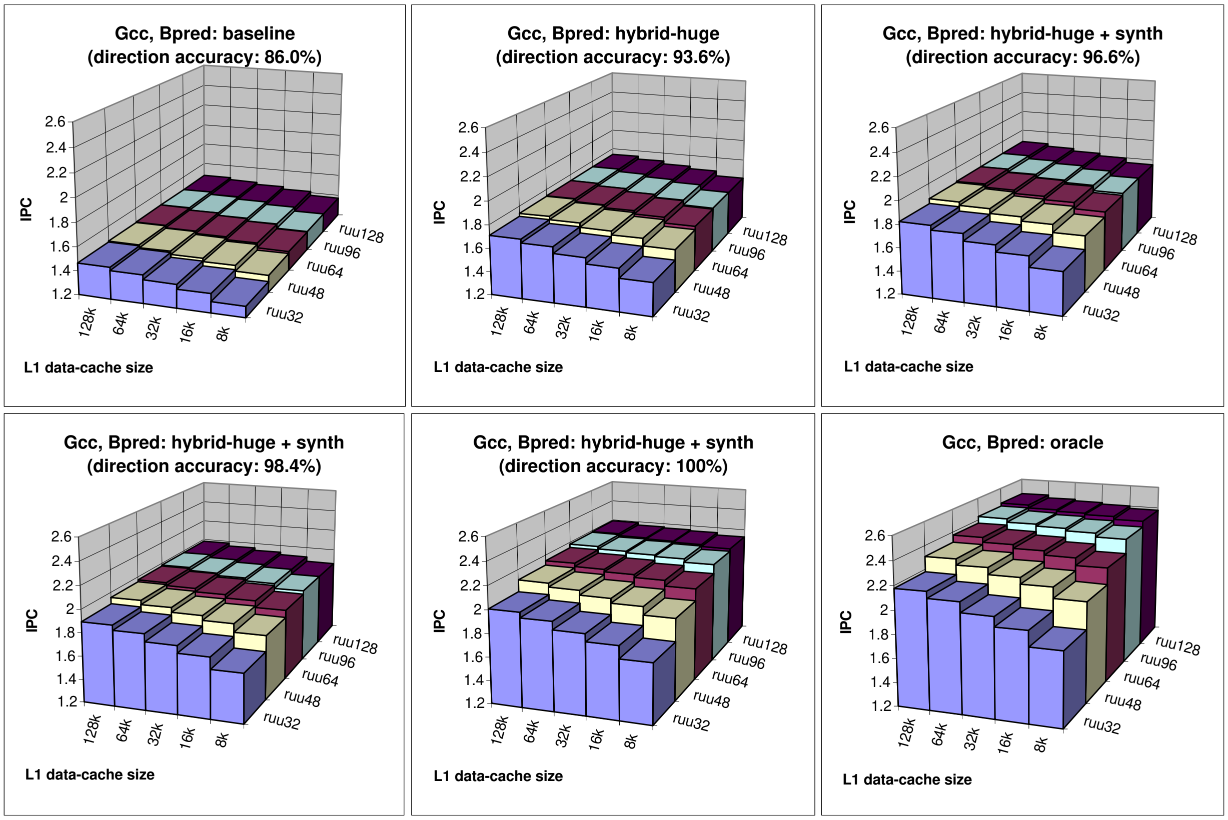

Fig. 9. Gcc's performance as a function of RUU size and L1 data-cache size for a variety of branch-prediction accuracies.

Figure 9 presents graphs of IPC vs. RUU size vs. L1 data-cache size for *gcc* over a range of direction-prediction accuracies. The prediction accuracies are indicated on each graph, and all the data are plotted on the same vertical scale. The first graph uses the baseline predictor and is taken from Figure 1; the second-to-last uses 100%-direction prediction and is taken from Figure 2; the intervening graphs use the mechanisms just described. The last graph in Figure 9 goes beyond 100%-direction accuracy by correcting all BTB misses, including those for returns and indirect branches—*i.e.*, oracle prediction. We present data for *gcc* as representative of the other benchmarks.

As *gcc*'s direction-prediction accuracy increases from 86% to 100%, not only does overall performance increase, but successive steps in RUU become useful (albeit slowly), contributing to the increase in IPC. Furthermore, as we mentioned earlier, with good branch prediction accuracy, bigger RUUs do such a good job of overlapping cache miss latencies that data cache size becomes less critical. The combination of these effects mean that the slopes along the RUU and cache axes are roughly equal at 98.4% accuracy. In fact, the slope along the cache axes flattens more slowly for *gcc* than for other benchmarks. Conversely, big caches miss so infrequently that big RUUs are not as necessary for hiding misses.

TABLE X

BRANCH RESOLUTION DELAYS FOR CONDITIONAL BRANCHES

	average	std-dev (n-1)		average	std-dev (n-1)
go	11.0401	5.4040	ijpeg	13.6127	8.4577
m88ksim	10.7539	3.9189	perl	13.0473	7.0491
gcc	11.4777	11.1734	vortex	24.0236	10.1844
compress	12.6343	11.3130	tomcatv	11.8892	6.8793
xlisp	12.7525	7.2824			

B. Return-Address-Stack Repair

The *return-address stack* is a small but important structure for achieving better control-flow prediction accuracy. Procedure returns present the same problem as other indirect branches: because a procedure might be called from many different locations (consider `printf()`), the target of a particular return instruction varies. Although general register-indirect jumps are hard to predict, the regular structure of call-return sequences permits a return-address stack to match returns with corresponding calls. Like other prediction techniques, the stack's prediction is only a hint: if the supplied return address is wrong, the misprediction is corrected in the writeback stage.

Return-address-stack accuracy can be an especially strong lever on performance. The stack is pushed and popped immediately after a call or return is fetched, *i.e.*, speculatively in the fetch stage, so some pushes or pops may correspond to wrong-path instructions. As with branch history, if the stack is not repaired, the wrong-path pops or pushes may corrupt it, as Jourdan *et al.* have pointed out [24]. They propose a sophisticated, self-checkpointing return-address stack that saves popped entries to avoid overwriting them with future mis-speculated pushes. So long as the stack does not overflow, this structure can return to any prior state.

We have shown that simply saving the current top-of-stack pointer at the time of each branch prediction, and restoring it after a misprediction, reduces by 50–93% return-address mispredictions from wrong-path corruption. Saving the top-of-stack contents along with the top-of-stack pointer virtually eliminates return-address mispredictions [50]. All the simulations in this paper have assumed pointer and top-of-stack contents repair.

C. Branch Resolution Delays

Even though a branch can move from decode to writeback in 5 cycles in our model, branches must wait for operands and then arbitrate for issue. In fact, branches typically take 10 or more cycles to resolve. Table X presents average branch-resolution times and their standard deviations. Note that the delay should be independent of the predictor organization, except as differences in prediction change the instruction flow through the processor.

Such long resolution times typically mean the processor has multiple branches in flight, in different stages of the processor. This requires a structure in which shadow register maps, return-address-stack patch state, and if applicable, branch-history fixup state are stored. The structure has a fixed depth—20 entries in the Alpha 21264 [26], for example—beyond which fetch stalls. For simplicity, however, our simulations have assumed unlimited shadow-state capacity.

VI. SIMULATION TECHNIQUES

The SPECint programs typically run for billions of instructions using reference inputs. Even smaller inputs typically run for hundreds of millions. But the fastest detailed microarchitecture simulators still take about an hour per 100 M instructions simulated on an UltraSPARC II or Pentium Pro. Simulating to completion is usually too expensive, especially for studies like this paper, which need hundreds of separate simulation runs.

Using very small inputs and scaling hardware structures accordingly is one possible solution. Scaling is risky, however: appropriate scaling factors are not always evident, and one must be careful to scale all relevant structures appropriately. Using smaller inputs may also change the relationship among various factors: for example, if various loops now iterate less often, branch-prediction behavior may change, inherent ILP may change, and so forth.

We focus on the obvious alternative: selecting a small, representative simulation window from a full-length run with the reference input. Many researchers do this because it makes simulation so much simpler. In particular, we simulate just 50 M instructions for each program. This section argues that a small simulation window like this must be chosen carefully, but can be reasonably representative of general program behavior. This makes it possible to simulate many different configurations or many benchmarks in a short period of time.

Sampling schemes are widely used in architecture studies and several pieces of prior work have investigated sampling methodologies, particularly related to cache memory simulations. For example, Laha *et al.* [30] studied the accuracy of memory reference trace sampling using caches that were 128 KB and smaller. Their study concluded that sampling allows accurate estimates of cache miss rates, but their results were presented for fairly unaggressive sampling techniques: they simulated 60% of all memory references.

As the fraction of references or instructions modeled becomes smaller, the question of how to "prime" the cache—how to deal with the unknown cache state at the beginning of each sample—becomes important. Our work takes a brute-force approach and simply simulates all instructions preceding the desired sample, just at a lower level of detail. Only the model's caches, branch predictor, and architectural state are updated. Other work has studied analytic models for estimating cache miss rates during the unprimed portion of the sample [25], [64], or described means for bounding errors by adjusting simulation lengths [34]. Iyengar and Trevillyan have derived the *R-metric* for measuring the representativeness of a trace [18], and they generate traces by scaling basic-block transition counts and adjusting selected instructions to optimize the R-metric. Their technique incorporates cache and TLB behavior as well as branch-prediction behavior, but because it uses traces, important mis-speculation effects may be omitted.

To accurately choose our simulation window, we have measured *interval* branch-misprediction rates for each of the benchmarks: *i.e.*, the misprediction rate computed separately over each million-instruction intervals in the program. This exposes representative segments of the trace. To illustrate this, Figure 10 shows the interval-branch-misprediction traces for four of the benchmarks, and pair of vertical lines delineates the 50 M-instruction segment we have chosen as our simulation window. Traces for all the programs appear in [54]. *M88ksim*'s trace resembles *perl*'s in terms of having a short initial phase and a fairly flat trace afterwards, and *Compress, vortex*, and *tomcatv* resemble *ijpeg* in terms of having clearly distinct, repeating phases. *Compress*'s phases, however, correspond to successive compress and decompress phases which are artifacts of the benchmark version. *Compress* also has a dramatic startup phase of about 1.7 billion instructions during which it generates the data to be compressed and decompressed. This is also an artifact of the benchmark version, but the benchmark versions are the ones many architects use for their simulations, and the misprediction rate during the startup phase is inordinately high (14.5% compared to a maximum of 11% for the rest of the program), creating a risk of substantially unrepresentative results if simulations include too much of this segment of *compress*'s execution.

We also obtained interval traces for data- and instruction-cache miss rates and ensured that our chosen segment was suitable with respect to these data as well. In most cases the different traces show similar qualitative behavior: major shifts most likely correspond to different phases of the programs' computations. Cache traces also appear in [54].

As mentioned with regard to *compress*, most programs show some sort of initial phase. In several cases the misprediction rate during initialization differs markedly from that during later phases of the program: *go, compress, perl*, and *tomcatv* are examples. If this initial behavior represents too large a fraction of the simulation, consequent results are unreliable. The simulation window should therefore be placed after any such initial phases. Fortunately, this can be done fairly quickly using fast-mode simulation. SimpleScalar will soon offer a checkpointing facility that removes even the need for fast-mode simulation [3]. Unless the initial behavior dominates the rest of the program's execution, its omission still gives reasonable results.

Although simulating a small but well-chosen window can produce representative results, many simulation-based studies have only modeled 50–100 M instructions from the beginning of a program's execution. This over-emphasizes initial behavior, and for some programs includes only initial behavior. This risks substantial distortion of results. To show how significantly results can change when the 50M-instruction simulation window is dominated by an initial phase, Figure 11 plots abbreviated versions of our 3D IPC vs. RUU size vs. data cache size graphs for *go, gcc*, and *perl*, for both regular and 100% branch prediction. The left two columns show data for the baseline branch prediction, and the right two columns show data for 100%-direction prediction. In each case, we compare a safe warmup period (we use the warmup times given in Table II) with just a million-cycle warmup period. In all cases, behavior between the two cases—between the first and second graphs, and between the third and fourth graphs—differs markedly, even though the one million cycles of warmup eliminate the most egregious startup effects. For *go* and *perl*, the initial phase has substantially different branch-prediction behavior, so such differences come as no surprise. *Gcc*, on the other hand, has no clear initial phase, but second-level cache misses contribute significantly, because 1 M is too short a warmup for big L2 caches. Of course, as simulations run longer, distortionary effects from initial phases have less impact on overall results. On the other hand, the initialization phase can be long: *e.g.*, 1.5 B instructions for *compress*, and over 2 B for *vortex* and many floating-point programs—prohibitively long for full-detail simulation.

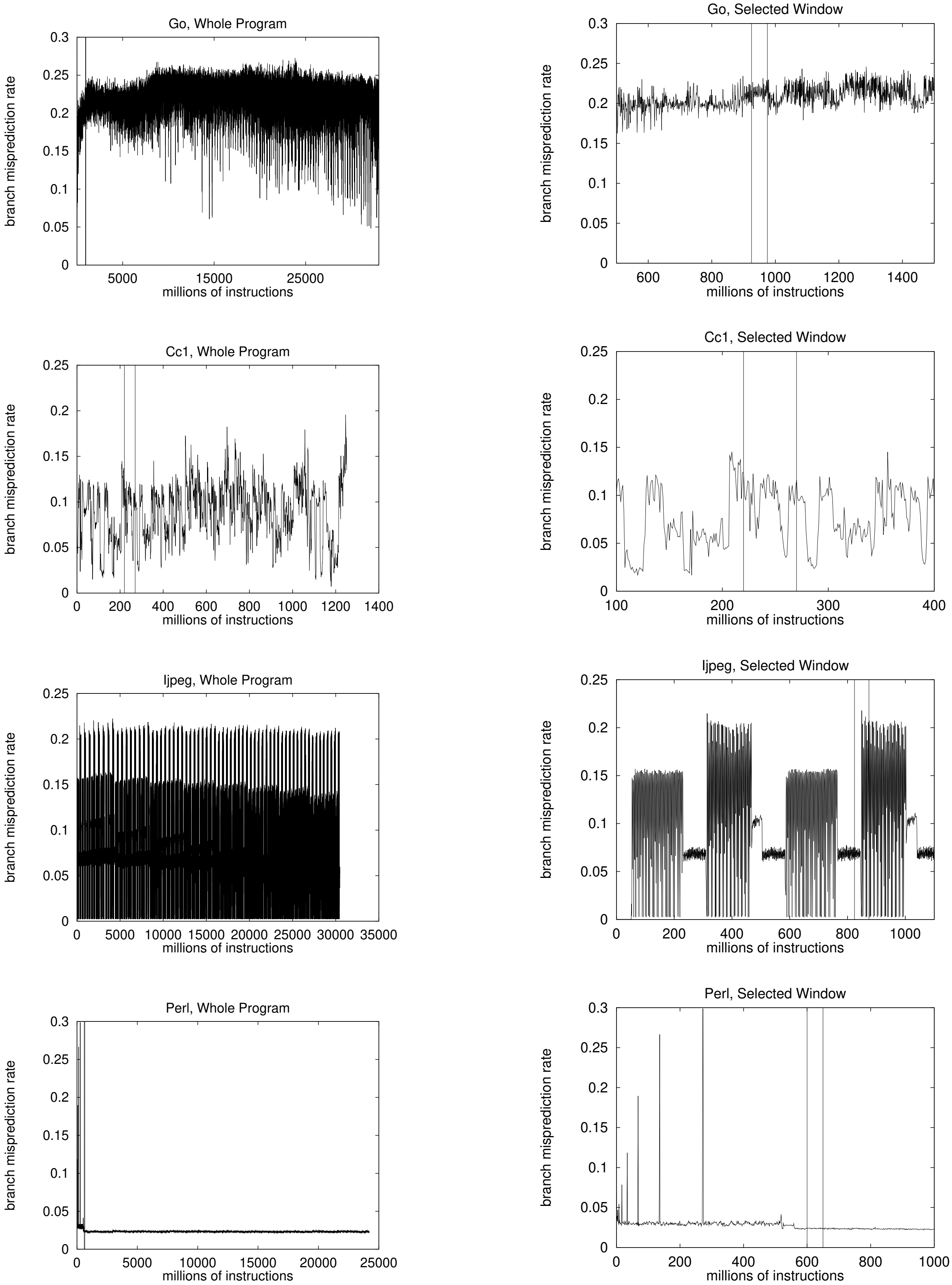

Fig. 10. Interval miss rates for go, gcc, ijpeg, and perl. The left-hand chart shows a trace from the entire program (using only the inputs described in Section II), and the right-hand chart shows a small segment of the trace. In both charts, two vertical lines indicate our simulation window (but in the left-hand charts, these vertical lines are sometimes too close to the x-axis.

Instead, a 50 M-instruction simulation window carefully chosen from later in the program's execution reliably gives representative results for SPECint programs. As evidence for this, Figure 12 gives further 3D graphs comparing our 50 M-window to a 250 M-instruction simulation window (we have 500 M-instruction data for go, so we present that instead). Again, compare the first graph to the second, and the third graph to the fourth. Particular IPC values change slightly, but the IPC-cache-RUU surfaces remain similar. Although data is not presented here, the same is true for all the other benchmarks (see [54] for the remaining data). The one slight exception is *gcc*, for which the 50 M step from an 8 K to a 16 K cache is not as pronounced as with 250 M. But data with a 100 M window matches the 250 M data. The discrepancy is due to L2 cache misses: with a perfect L2, the 50 M and 250 M data match closely. *Ijpeg* is also sensitive to L2 misses, because the L2 miss behavior does not settle down until several execution phases have passed. But we did find a 50 M-instruction window that captures representative behavior.

Gathering the data for Figure 12 forced us to refine our choice of window in the case of *ijpeg* and *gcc*. Choosing a representative 50 M-window window using the interval traces is a good heuristic, but verifying it by comparing 50 M-instruction plots IPC vs. RUU size vs. D-cache size against 250 M-instruction plots provides even more reliable simulation windows.

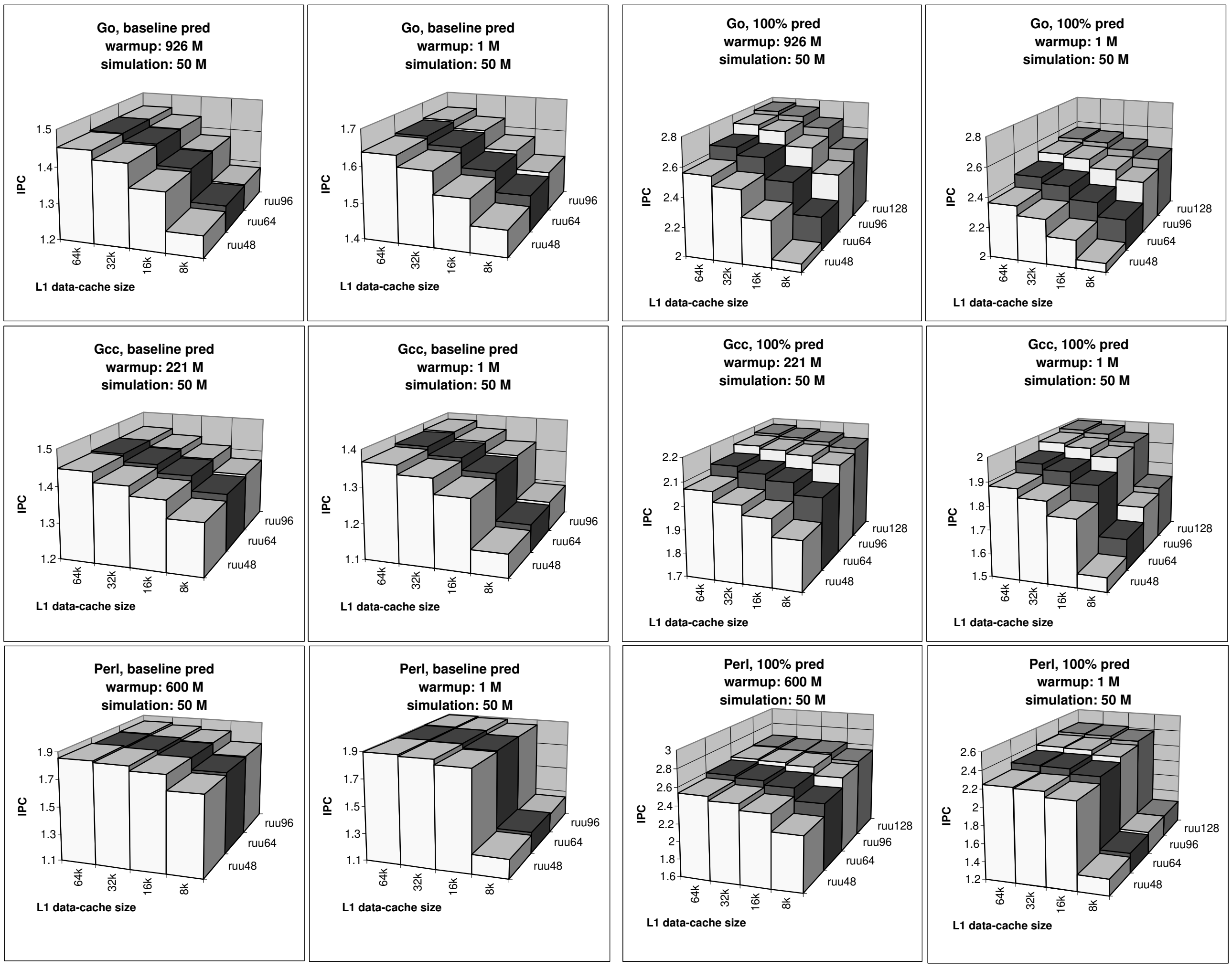

Fig. 11. Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations and no simulator warmup vs. our chosen warmup.

VII. Related Work

Latency tolerance and branch prediction have been studied extensively as separate architectural issues and have each been considered with respect to their interactions with ILP. We give a sampling of relevant work here. Some

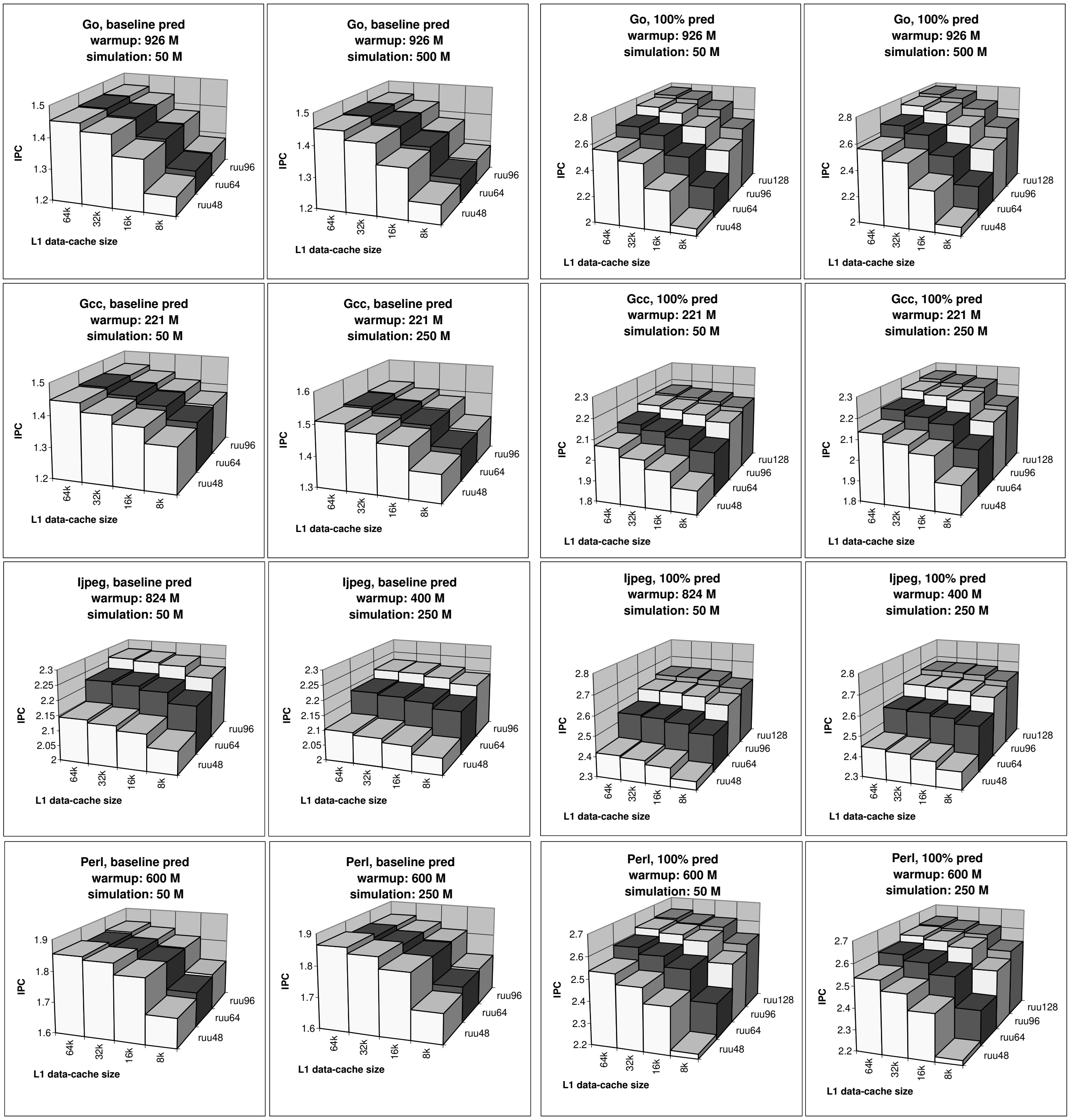

Fig. 12. Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations vs. 250 M-instruction simulations (or in the case of go, 500 M-instruction simulations).

of the most basic work in the field of superscalar processors has focused on identifying the limits of ILP in different applications. For example, Wall and Jouppi have examined this issue for a variety of realistic and idealized machine configurations [23], [60]. Other work by Smith, Johnson, and Horowitz has also explored the limits on instruction issue afforded by a suite of integer and scalar floating point applications [55]. These works have focused on inherent parallelism in the application. Lam and Wilson have explored the impact of control flow and showed how relaxing control dependence constraints potentially improves performance [31]. Woo *et al.* have characterized the behavior of the SPLASH-2 suite in terms of sensitivity to various parameters and discussed methodological issues for simulating parallel applications [63]. Maynard *et al.* have shown that results using any particular suite of benchmarks must be interpreted with care. For example, SPEC programs have small text sizes, execute little system code, and execute minimal random I/O, while many commercial programs have large text sizes, execute a substantial amount of system code, and perform a substantial amount of random I/O [35]. Many commercial programs also have a large static branch footprint, while most SPEC programs do not [11].

Other fundamental research has focused on understanding and improving branch prediction accuracy via both hardware and software means. Lee, Chen, and Mudge [32], Sechrest, Lee, and Mudge [49], and Skadron *et al.* [53], among others, have described and measured the different causes of mispredictions, and Evers *et al.* [13] recently measured the benefits of branch correlation. Work by Yeh and Patt [65], Pan *et al.* [40], McFarling [36], Sprangle *et al.* [57], and Eden and Mudge [10], among others, have proposed hardware mechanisms for keeping multi-level branch predictors, for tracking correlations between branches, and for avoiding contention among branches in the predictor's state tables. Such hardware branch prediction mechanisms have been widely incorporated into commercial designs [16], [26], [38]. Some work has also explored software-based branch-prediction techniques: Young, Gloy, and Smith [66], [67] have demonstrated compiler-based methods for correlated branch prediction, while Mahlke and Natarajan [33] and August *et al.* [2] have examined branch prediction synthesized in the compiler. Rotenberg *et al.* take an even more aggressive tack: they reorganize the processor around *traces*, groups of basic blocks which have been coalesced into a single unit. When the fetch engine hits in the trace cache, it can provide several basic blocks every cycle without the need for merging cache blocks, multiple-branch or multi-ported predictors, or a multi-ported instruction cache [19], [46]. On a more theoretical level, Gloy and Emer have developed a general language for describing predictors, and they show how it can be used for automated synthesis of predictors. Resulting structures can be complex, but this model may yield insight into avenues for further improvement [12].

Predicting branch targets is important, too. To better understand the performance effect of BTB misses, Michaud, Seznec, and Uhlig measure compulsory BTB misses [37] for all types of branches. Calder and Grunwald [6], Chang, Hao, and Patt [8], and Driesen and Hölzle [9] have all examined ways to augment the BTB by taking prior branch-target history into account. None of these papers explicitly treat predicting return-instruction targets, for which return-address stacks can virtually eliminate mispredictions. Jourdan *et al.* [24] and Skadron *et al.* [50] both focus on return-address-stack design, especially on mechanisms for repairing the return-address-stack after it has been modified by mis-speculated instructions.

While branch prediction is a well-known performance "lever," its relationship to cache design decisions has not previously been quantitatively evaluated. Jouppi and Ranganathan find in [22] that branch prediction is a stronger limitation on performance than memory latency or bandwidth.

Finally, cache design has been a key issue with processor architects for several years now. Many papers study the tradeoffs between L1 cache size and speed; the most recent, simulating a MIPS R10000 model, is by Wilson and Olukotun [62]. Prefetching helps tolerate load latencies, but under OOE must take into account that many misses are adequately tolerated without prefetching. Most techniques [39], [42] were developed with simpler processor models in mind, but [48], for example, discusses data prefetching for the HP PA-8000. Farkas *et al.* [14] have recently provided some insights regarding memory system design for dynamically-scheduled processors, and Johnson and Hwu [21] discuss a cache allocation mechanism to prevent rarely accessed data from displacing frequently accessed lines. Srinivasan and Lebeck measure loads' latency tolerance and demonstrate the importance of quickly completing loads that feed branch instructions [58]. These papers do not touch on the relationship of branch prediction to cache design as our paper does.

VIII. Conclusions

By presenting a database of simulation results for the SPECint programs, this paper examines shifting tradeoffs among instruction-window size, first-level data- and instruction-cache size, and branch-prediction accuracy in high-performance processors. The results show that regardless of cache size, having more than 48 RUU (instruction window) entries yields almost no benefit to performance for many benchmarks. This mostly occurs because deeper entries are not used: mispredictions occur often enough to prevent the RUU from building up a large pool of instructions. Even when the deeper entries are active, they usually contain only mis-speculated instructions. This falls far short of the optimum: for programs with branch prediction accuracies near 100%, or if branch prediction could be made perfect, adding RUU even out to 256 entries yields strong benefits.

For many SPECint programs, L1 data-cache size is a strong lever on performance. This becomes less so as branch prediction improves and deeper RUU entries become useful, because the deeper RUU affords enough lookahead to overlap L1 misses with useful computation. At the extreme of 100% prediction accuracy, data cache size hardly matters at all. Conversely, as cache size increases, bigger RUUs help less because fewer misses occur and less lookahead is necessary. Nevertheless, as branch-prediction accuracy improves, sensitivity to RUU size increases quickly and RUU effects eventually dwarf L1 data-cache effects.

The picture is somewhat different for L1 instruction-cache misses, which are typically so tightly clustered—often only 1 or 2 cycles elapse between misses—that even with 100% prediction and a deep window of instructions, a too-small I-cache is the dominant bottleneck.

The most important bottleneck nevertheless remains branch prediction. L1 data-miss penalties will always inspire innovations, but caches are now becoming sufficiently big and sophisticated that future work should perhaps focus specifically on latency-intolerant misses and on better branch prediction. As architects attack the branch-prediction bottleneck with more sophisticated hardware schemes and alternative techniques—trace caches [45], compiler-enhanced hardware prediction [2], [33], predication [20], [44], and multi-path execution [1], [27], [28], [61]—larger RUUs will become attractive.

This paper also considers sampling techniques to allow shorter but full-detail simulations. For the SPECint programs, fairly short samples of 50 M instructions from simulations with reference inputs yield good results, but accuracy becomes quite sensitive to the choice of the simulation window. The sample must come from a point after any initial execution phases, which can be quite long, up to several billion instructions.

Because latencies can overlap or compound each other in modern, out-of-order processors, design parameters interact in sometimes complex ways. This paper has illustrated a number of resulting tradeoffs, but more importantly, has highlighted some potential pitfalls that can result from unwise combinations of branch-prediction, instruction-window, and cache configurations. The comprehensive data presented here has two main contributions. First, it quantitatively shows the importance of considering these configuration issues in conjunction, rather than choosing sizes individually or independently. Second, this paper helps cull the design space to avoid expensive or methodologically flawed simulations.

Acknowledgments

This work was supported in part by NSF grant CCR-94-23123, NSF Career Award CCR-95-02516 (Martonosi), and an NDSEG Graduate Fellowship (Skadron). We thank Kai Li and the referees for helpful feedback in preparing this paper.

References

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multi-path execution: Opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing*, pages 101–08, July 1998.

[2] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 84–93, Feb. 1997.

[3] D. Burger. Personal communication, Mar. 1998.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

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

[6] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Jan. 1994.

[7] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 252–57, Dec. 1995.

[8] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 274–83, June 1997.

[9] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167–78, July 1998.

[10] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–77, Dec. 1998.

[11] J. Emer. Personal communication, June 1997.

[12] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 304–14, June 1997.

[13] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, June 1998.

[14] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–43, May 1997.

[15] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proccedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Oct. 1992.

[16] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, Feb. 16, 1995.

[17] E. Hao, P.-Y. Chang, and Y. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, Nov. 1994.

[18] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. IBM Research Report RC 20610, Oct. 1996.

[19] Q. Jacobsen, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23, Dec. 1997.

[20] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 100–13, Dec. 1996.

[21] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–26, June 1997.

[22] N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *The Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997. http://ayer.CS.Berkeley.EDU/isca97-workshop.

[23] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proccedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–82, Apr. 1989.

[24] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal on Parallel Programming*, 25(5):363–83, Oct. 1997.

[25] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. Technical Report 1048, Univ. of Wisconsin Computer Sciences Department, Sept. 1991.

[26] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design*, Oct. 1998.

[27] A. Klauser, V. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–59, July 1998.

[28] R. Kol and R. Ginosaur. Kin: A high performance asynchronous processor architecture. In *Proceedings of the 12th International Conference on Supercomputing*, pages 433–440, July 1998.

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

[30] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, pages 1325–1336, Nov. 1988.

[31] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[32] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 4–13, Dec. 1997.

[33] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–164, Dec. 1996.

[34] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurment and Modeling of Computer Systems*, pages 248–59, May 1993.

[35] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting charactersitics and cache performance of technical and multi-user commercial workloads. In *Proccedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, Oct. 1994.

[36] S. McFarling. Combining branch predictors. Technical Note TN-36, DEC WRL, June 1993.

[37] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 292–303, June 1997.

[38] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, Jun. 1995. Version 1.0.

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

[40] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proccedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct. 1992.

[41] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 165–75, Dec. 1996.

[42] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice Univ., May 1989.

[43] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[44] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, pages 12–35, Jan. 1989.

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

[46] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

[47] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[48] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–73, June 1997.

[49] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, May 1995.

[50] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–71, Dec. 1998.

[51] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 144–155, Feb. 1997.
[52] K. Skadron, D. W. Clark, and M. Martonosi. Speculative updates of local and global branch history: A quantitative analysis. Technical Report TR-589-98, Princeton Dept. of Computer Science, Dec. 1998.
[53] K. Skadron, M. Martonosi, and D. W. Clark. Alloying global and local branch history: Taxonomy, performance, and analysis. Technical Report TR-594-99, Princeton Dept. of Computer Science, Jan. 1999.
[54] K. Skadron, M. Martonosi, and D. W. Clark. Selecting a single, representative sample for accurate simulation of specint benchmarks. Technical Report TR-595-99, Princeton Dept. of Computer Science, Jan. 1999.
[55] M. D. Smith, M. Johnson, and M. Horowitz. Limits on Multiple Instruction Issue. In *Proccedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Apr. 1989.
[56] G. S. Sohi and A. S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.
[57] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 284–91, June 1997.
[58] S. Srinivasan and A. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 148–59, Dec. 1998.
[59] The Standard Performance Evaluation Corporation. WWW Site. http://www.specbench.org, Dec. 1996.
[60] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proccedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–88, Apr. 1991.
[61] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–49, July 1998.
[62] K. M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 121–32, June 1997.
[63] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.
[64] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.
[65] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–66, May 1993.
[66] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–86, June 1995.
[67] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proccedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–41, Oct. 1994.

Kevin Skadron received a BS in electrical and computer engineering and a BA in economics from Rice University in 1994, and will complete his Ph.D. in computer science in the summer of 1999 at Princeton University. His research interests lie in the area of microarchitecture—branch prediction, memory hierarchies, and complexity/performance tradeoffs—as well as in the areas of compiler-architecture interactions, simulation methodology, and operating systems.

Pritpal S. Ahuja received a BS in computer science from McGill University in 1993, and a MA in computer science from Princeton University in 1996. He is currently pursuing his PhD in computer science from Princeton University. His research interests lie in computer architecture. is a Ph.D. candidate in computer science

Margaret Martonosi is an Assistant Professor of Electrical Engineering at Princeton University. Her research interests include both architecture and applications issues in high-performance computers. In particular, her recent NSF- and DARPA-sponsored research focuses on applying performance analysis techniques to program customization and dynamic adaptive computation. Martonosi received her Ph.D. from Stanford University in 1993, and also holds an M.S. from Stanford University and a B.S. from Cornell University, all in Electrical Engineering.

Douglas W. Clark received the B.S. degree in engineering and applied science from Yale University in 1972 and the Ph.D. degree in computer science from Carnegie-Mellon University in 1976.

He worked at Xerox PARC for four years before joining Digital Equipment Corporation in 1980. At Digital he contributed to the design and performance analysis of several VAX processors. In 1993 he joined the faculty of Princeton University as Professor of Computer Science. At Princeton he has done work in microarchitecture and in performance monitoring.