Selecting a Single, Representative Sample for Accurate Simulation of SPECint Benchmarks

Technical Report TR-595-99

Kevin Skadron, Margaret Martonosi, Douglas W. Clark
Depts. of Computer Science and Electrical Engineering
Princeton University
{skadron,doug}@cs.princeton.edu, martonosi@ee.princeton.edu

Abstract

Detailed micro-architecture studies often require time-consuming, cycle-accurate simulation. Unfortunately, many benchmarks of interest, like the SPEC benchmarks, run for tens of billions of instructions and are prohibitive to simulate to completion. Instead, researchers typically must confine themselves to simulating samples of the program's overall execution. This report presents detailed *interval-trace* data about the SPECint programs' branch-prediction and cache behavior. The report then argues that accurate simulation results can be obtained using a single, short simulation window of just 50 million instructions. They key to obtaining accurate results is to choose this simulation window carefully, and in particular to avoid unrepresentative behavior at the beginning of programs' execution.

1 Introduction

The SPECint programs [23] typically run for billions of instructions using reference inputs. Even smaller inputs typically run for hundreds of millions. But the fastest, detailed, cycle-accurate microarchitecture simulators still take about an hour to simulate 100 million instructions on an UltraSPARC II or Pentium Pro. Simulating to completion is therefore usually too expensive, especially for architecture studies that explore a parameter space and 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.

This work describes a simple alternative: selecting a small, representative simulation window from a full-length run with the reference input.

Many researchers use a single, small, simulation window because it makes simulation so much simpler. This report argues that short simulation windows of just 50 million (M) instructions can indeed suffice to obtain results that are representative of the program's overall behavior. This makes it possible to simulate many different configurations or many benchmarks in a short period of time.

The key to making this technique work is to choose the simulation window carefully. To make this possible, we have gathered *interval-trace* data about the SPECint programs' branch-prediction and cache behavior. These interval-trace data report the branch-prediction and cache performance over each one-million-instruction *interval* in the program. This provides a localized picture of the program's behavior that makes it possible to identify representative segments that might be candidates for the simulation window.

The most important piece of information provided by the interval-trace data is an indication of when programs enter new phases of execution in which behavior is dramatically different. In particular, most programs have a startup phase with distinctive behavior that is unrepresentative of the rest of the execution's behavior. Unfortunately, many studies using a single, short simulation window simply take 50 M, 100 M, or 250 M instructions from the beginning of the program's execution, and the last section of this report shows how badly flawed that approach can be. That section also uses results from long cycle-level simulations to show that a properly placed short simulation window does provide representative behavior.

The rest of this report is organized as follows. The next section describes related work, and the following section describes our simulator and benchmarks. Then Section 4 presents our interval-trace data and indicates the best 50 M instruction simulation windows we have found. Section 5 uses cycle-level simulations to verify these simulation windows, and Section 6 concludes the report.

2 Related Work

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.* [12] studied the accuracy of memory reference trace sampling using caches that were 128 Kbytes and smaller. Their study concluded that sampling allows accurate estimates of cache miss rates, but their results were presented for fairly unaggressive sampling techniques that simulate 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. This 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 [9, 26], or described means for bounding errors by adjusting simulation lengths [13]. Iyengar and Trevillyan have derived the *R-metric* for measuring the representativeness of a trace [7], 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.

This report extends the data presented in Skadron's thesis [17] and in the paper by Skadron *et al.* [18].

3 Simulation Methodology

This work uses two simulators. The first is an instruction-level simulator used to obtain the interval-trace data, and the second is a cycle-accurate simulator used to verify the representativeness of our chosen simulation windows.

Both simulators are based on the SimpleScalar toolset [2]. 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. Both simulators interpret executables compiled by *gcc* version 2.6.3 for a virtual instruction set called the "PISA" that most closely resembles MIPS IV [16].

3.1 Simulator

3.1.1 Interval-trace simulations

The simulator that obtains the interval-trace data is based on SimpleScalar 2.0's instruction-level simulators. Our interval simulator performs conventional instruction-level simulation, modeling each instruction's behavior in the branch predictor and cache modules, but every 1 M cycles it stops and reports the branch-misprediction and cache-miss rates for the previous 1 M-instruction interval, and then it resets these statistics in preparation for the next interval. At the end of the simulation, these interval miss rates are written to a file which can then be plotted.

3.1.2 Cycle-accurate simulations

Cycle-accurate simulations use *HydraScalar*, our heavily modified version of SimpleScalar's *sim-outorder* simulator. 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 is 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), five-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, writeback, and commit. An arbitrary number of further stages can be added between decode and issue to simulate time spent renaming and enqueuing instructions (the experiments in this work add three extra cycles). Issue selects the oldest ready instructions.

Table 1 summarizes the baseline model used for most of this work, which loosely resembles the configuration of an Alpha 21264 [10]. The simulations use a two-level, non-blocking cache hierarchy with miss-status holding registers (MSHRs) in both the first-level (L1) and second-level (L2) caches [11]. 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, especially for non-numeric codes [20]. HydraScalar simulates a unified active list, issue queue, and rename register file. This type of instruction window is called a *register update unit* or RUU [22]. 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. A load-store queue (LSQ) disambiguates memory references: stores may only pass preceding memory references whose addresses are known not to conflict. This work does not explore LSQ size, so for simplicity the LSQ is fixed at one-half the RUU size to eliminate artifacts from stalls should the LSQ fill up.

The branch predictor consists of a hybrid, McFarling-style predictor [4, 15] for predicting the direction of conditional branches, and a branch target buffer (BTB) for predicting the target address of both conditional and unconditional branches. Many entries in the PHT (the direction-predicting pattern history table consisting of the saturating, two-bit counters) correspond to not-taken branches (or are simply idle). For this reason, an entry is only allocated in the BTB for taken branches. This permits the BTB to have fewer entries than the PHT [3]. The fetch stage modeled by HydraScalar 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.

HydraScalar updates the predictor state during the instruction-commit stage, except where noted. This means there is a window of time, while a branch traverses the pipeline, during which its outcome is not available and the branch predictor uses slightly "stale" state. The resulting prediction accuracies 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 only a recent innovation because it requires fixup mechanisms to undo corruption from incorrect updates. The importance of speculative update is discussed in [6], and mechanisms for implementing fixup appear in [8] and [21]. The only known example of speculative update appears in the Alpha 21264, which implements speculative update with fixup for just the global-history portion of its hybrid predictor [10].

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 (it only remembers the last target). 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. HydraScalar models a simple mechanism, described in [19], that does not guarantee restoration of the correct state, but in practice virtually eliminates corruption. All the simulations reported here use this mechanism and, like the Alpha 21264, use a 32-entry stack [10].

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

Parameter	Value	Comments
	Processor core	
RUU size	64	Instruction window
LSQ size	32	Enforces load-store ordering
Decode latency	4 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	2	any combination of loads or stores
	Branch prediction	
Cond. branch predictor	Hybrid: 4K entry x 12-bit GAg and 1K entry x 10-bit x 1K entry PAg selector: 4K entry x 12-bit	(component #1) (component #2) (like GAg)
BTB	2048-entry, 2-way	updated only if taken
Return-addr. stack	32-entry	repaired after mis-speculation
Mispredict penalty	7 cycles	2 cycles for BTB-miss only
	Memory hierarchy	
L1 D-cache	64 K, 2-way (LRU), 32 B blocks	8 MSHRs, 1-cycle latency
L1 I-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

Table 1: Baseline configuration simulated by HydraScalar.

detected early. A BTB miss therefore only experiences a two-cycle penalty. Indirect jumps need to read the register file, which is at best expensive to do from decode and is therefore not supported in these models. When the BTB mispredicts these targets, the error is only detected in the writeback stage.

3.2 Benchmarks

This research focuses on the SPEC95 integer benchmarks [23], summarized in Tables 2 and 3, and uses the provided SPEC95 "reference" inputs. Like any other suite, SPEC has its shortcomings: for example, many of the programs fit in small instruction caches. Because commercial workloads differ in some ways from the SPECint programs [14], users should exercise care in translating this report's results to such programs: the trends and pitfalls identified here should hold true for many programs, but clearly not for all. Another possible pitfall is that this work excludes operating-system calls, because SimpleScalar does not not support that and because it adds substantially to simulation times. Some program behavior—including branch behavior—does vary when operating system calls are included [5]. Fortunately, this is not a problem for most SPEC benchmarks, as they call the operating system only rarely [5]. Furthermore, the simulation lengths used here are short and on the order of context-switch intervals. The next section describes how these short simulation windows are selected.

The integer benchmarks are the primary focus of this work, because they are the subject of so many computer architecture studies. The same methodology could easily be extended to the SPECfp programs. 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 only one has been chosen. *Go* is run with a playing level of 50 and a 21x21 board with the `9stone21` input. For *m88ksim*, the `dhrystone` input is used; for *gcc*, `cccp.i`; for *ijpeg*, `vigo.ppm`; and for *perl*, the scrabble game is used. Unlike these benchmarks, *xlisp* is run with all the supplied LISP files as arguments, because each individual file is small.

4 Using Interval Traces to Select the Simulation Window

To accurately choose the simulation window, we have first measured interval branch misprediction rates for each of the benchmarks: *i.e.*, the misprediction rate computed separately over each million-instruction interval in the program. (Note that by "misprediction rate" we mean the rate at which the direction of conditional branches is mispredicted.) This helps expose small but representative segments of the programs, while smoothing out the finest-grain variations in behavior. To illustrate this, Figures 1–3 show the interval-branch-misprediction traces for the benchmarks, on which a pair of vertical lines delineates the 50 M instruction segment chosen as the simulation window. Note that *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. In fact, the misprediction rate during the startup phase is inordinately high—14.5% compared to a maximum of 11% for the rest of the program—and this creates a risk of substantially unrepresentative results if simulations include too much of this segment of *compress*'s execution relative to actual compress/decompress behavior.

Table 4 shows the 50 M instruction windows that we chose, and that are evaluated in Section 5.

Interval traces were also obtained for data and instruction cache miss rates to ensure that the chosen segment was suitable with respect to cache as well as branch-prediciton behavior. The different types of traces show similar qualitative behavior, with major shifts typically occurring at the same point in each type of trace for a particular benchmark. Cache traces appear in Figures 4–12, grouped by benchmark. First-level caches are 32 Kbytes, 4-way associative, with 32 byte lines. Second-level (L2) caches are 1 Mbyte, direct-mapped, with 32 byte lines. Due to an error in our simulation scripts, for some programs we have data for unified L2 caches, for others we have data for separate instruction- and data- L2 caches.

	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	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

Table 2: Benchmark summary—branch statistics.
Statistics are taken only from the post-warmup, 50 M-committed-instruction simulation window, and use the baseline configuration in Table 1. "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.

	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

Table 3: Benchmark summary—cache statistics.
L2 cache miss rates are global, that is, over all references made by the program.

go	926–976 M	ijpeg	824–874 M
m88ksim	26–76 M	perl	601–651 M
gcc	221–271 M	vortex	2451–2501 M
compress	2576–2626 M	tomcatv	2276–2326 M
xlisp	271–321 M		

Table 4: Selected simulation window.
(millions of instructions)

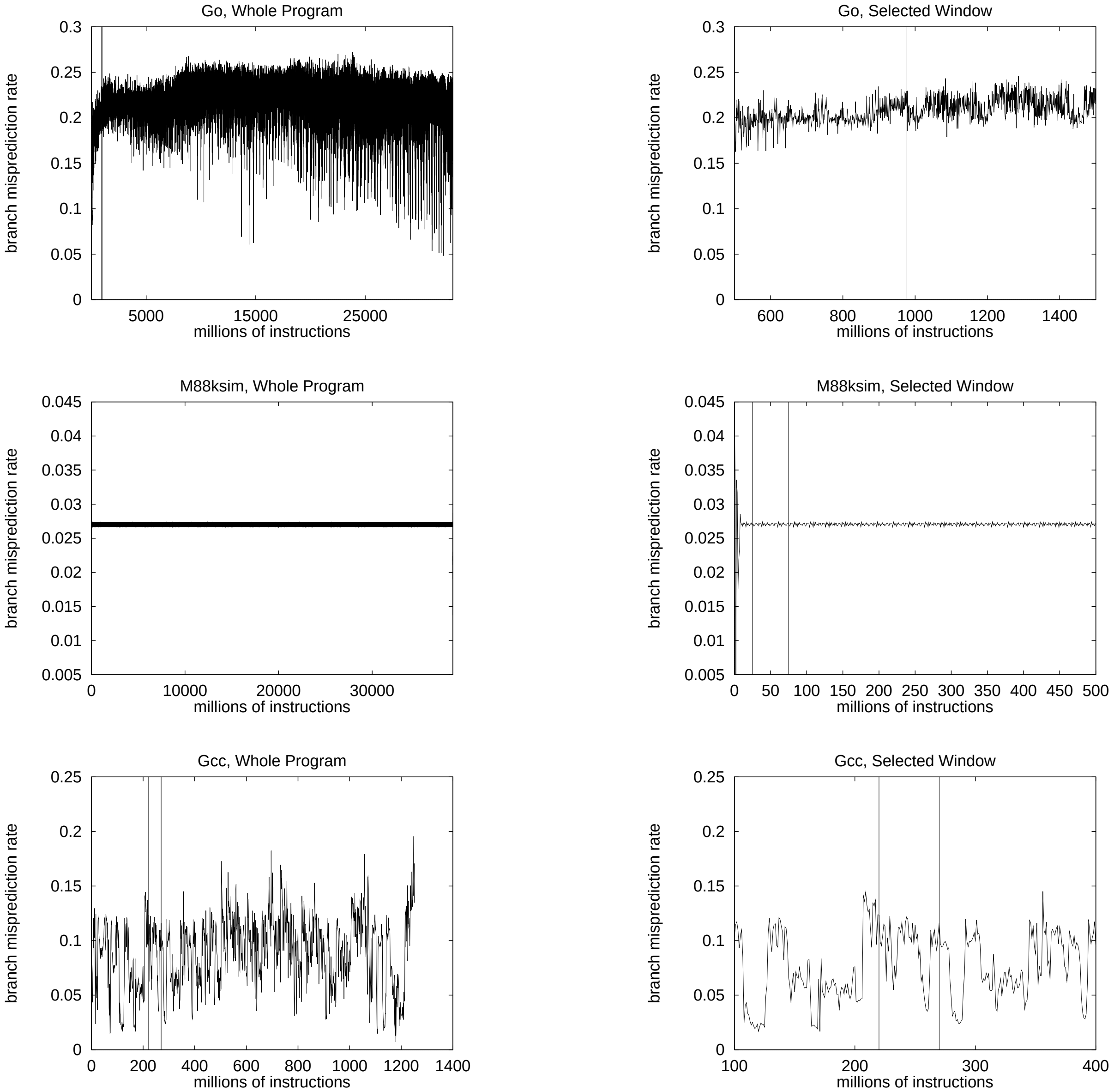

Figure 1: Interval misprediction rates for go, m88ksim, and gcc.
The left-hand chart shows a trace from the entire program (using only the inputs described in Section 3.2), and the right-hand chart shows a small segment of the trace. In both charts, two vertical lines indicate the chosen simulation window (but in the left-hand charts, these vertical lines are sometimes too close to the x-axis.

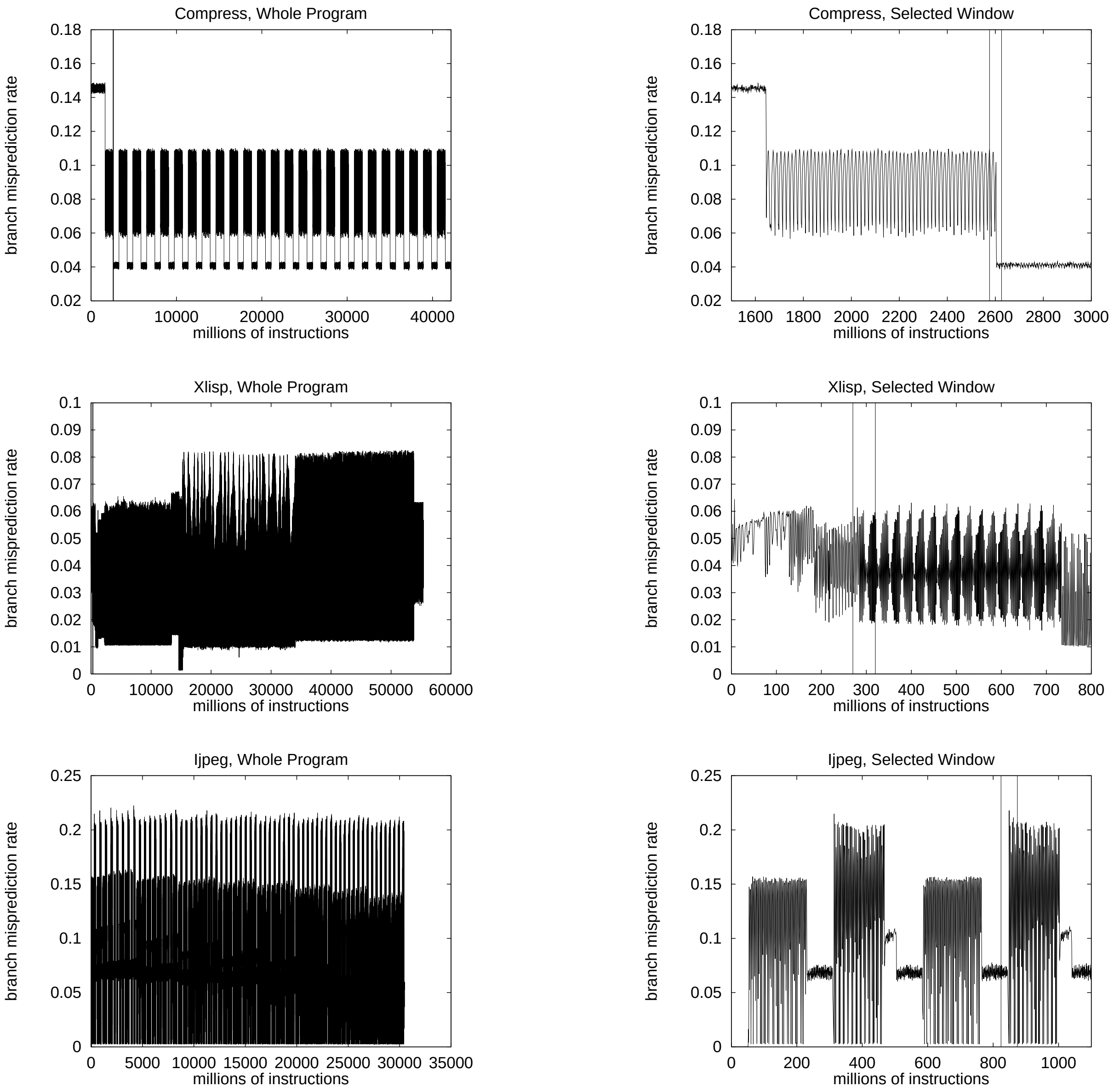

Figure 2: Interval misprediction rates for compress, xlisp, and ijpeg.

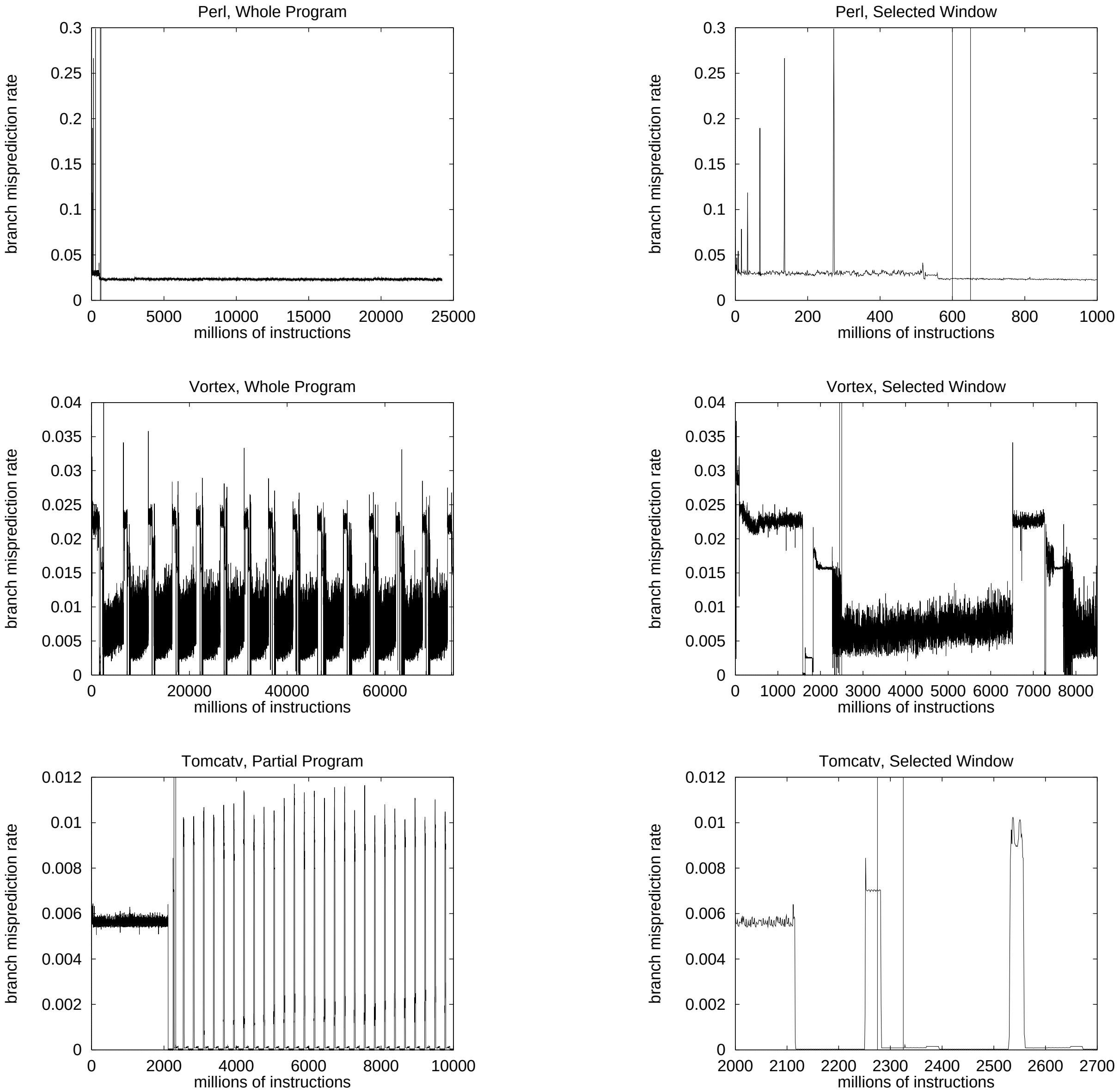

Figure 3: Interval misprediction rates for perl, vortex, and tomcatv.
Tomcatv with reference data runs for so long that a chart for the entire program does not show useful detail. Instead, the left-hand chart shows a trace for the first 10 billion instructions.

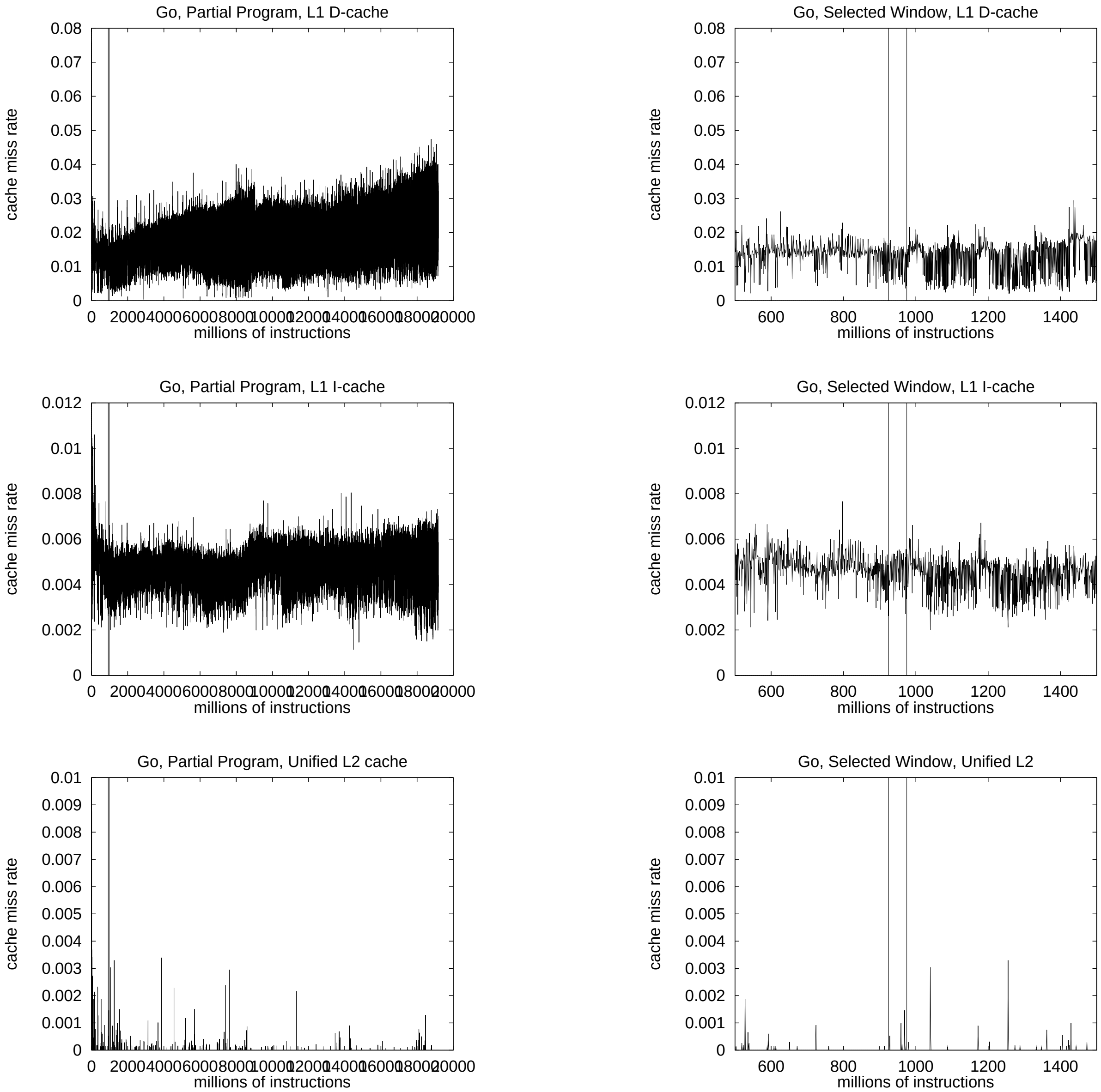

Figure 4: Interval cache miss rates for go. Unified L2.

As before, the left-hand chart shows a trace from the entire program (using only the inputs described in Section 3.2), and the right-hand chart shows a small segment of the trace. In both charts, two vertical lines indicate the chosen simulation window (but in the left-hand charts, these vertical lines are sometimes too close to the x-axis. In the case of go, we have only collected cache data for the first 19 billion instructions, even though the program runs about 30 billion instructions.

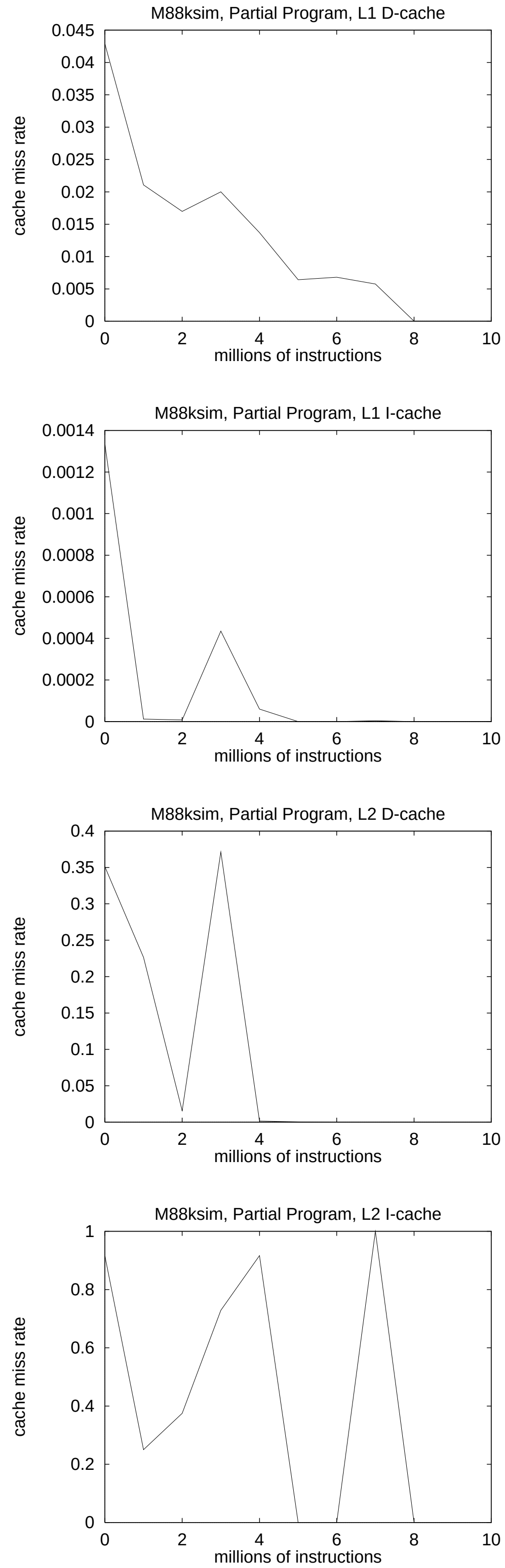

Figure 5: Interval cache miss rates for m88ksim. M88ksim fits into all the caches, so after the first 10 million instructions, it never misses. (Actually, the L1 D-cache sustains a consistent miss rate of about 0.00002.) For this reason, we just show the trace for the first 10 data points. The program actually runs for nearly 40 billion instructions.

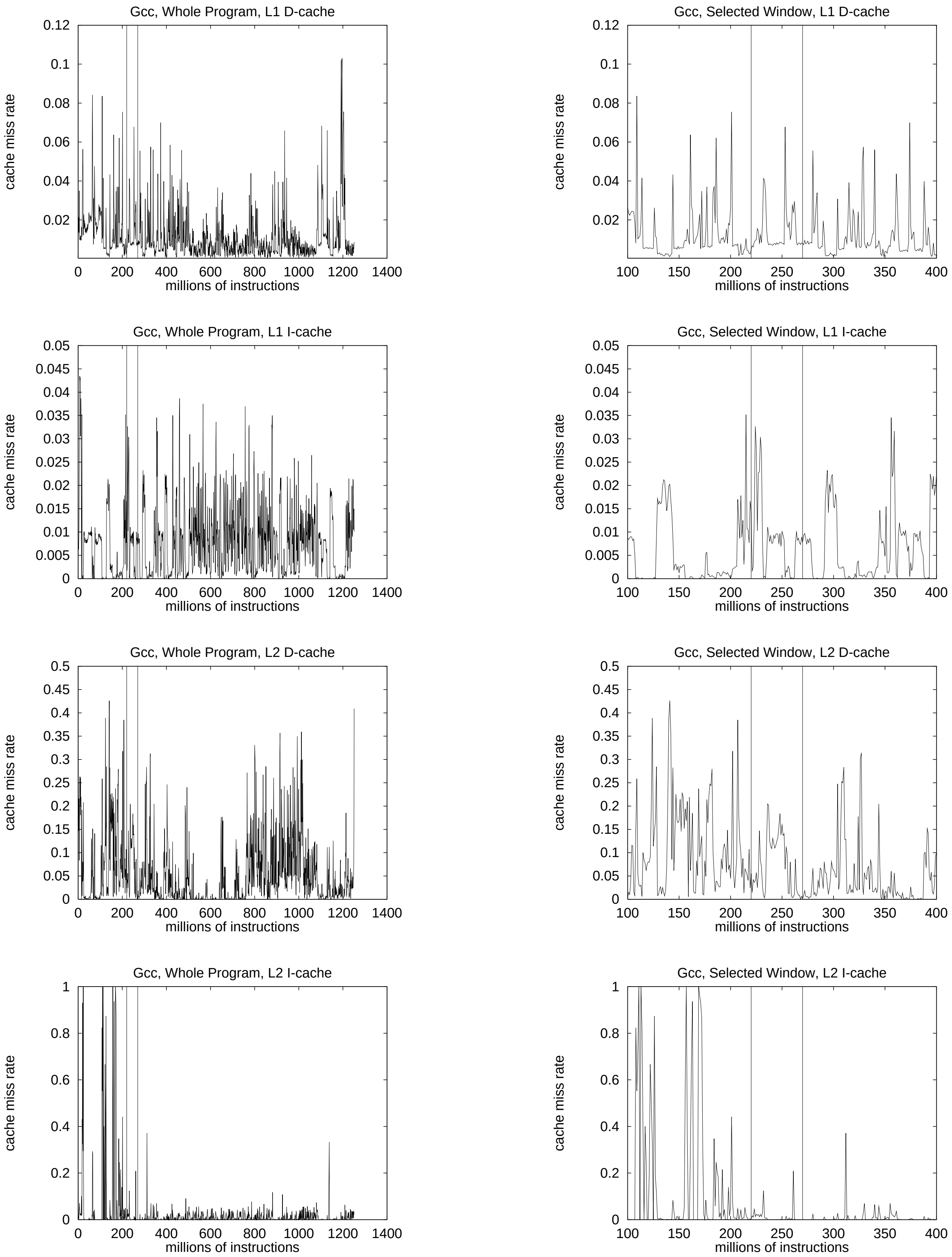

Figure 6: Interval cache miss rates for gcc.

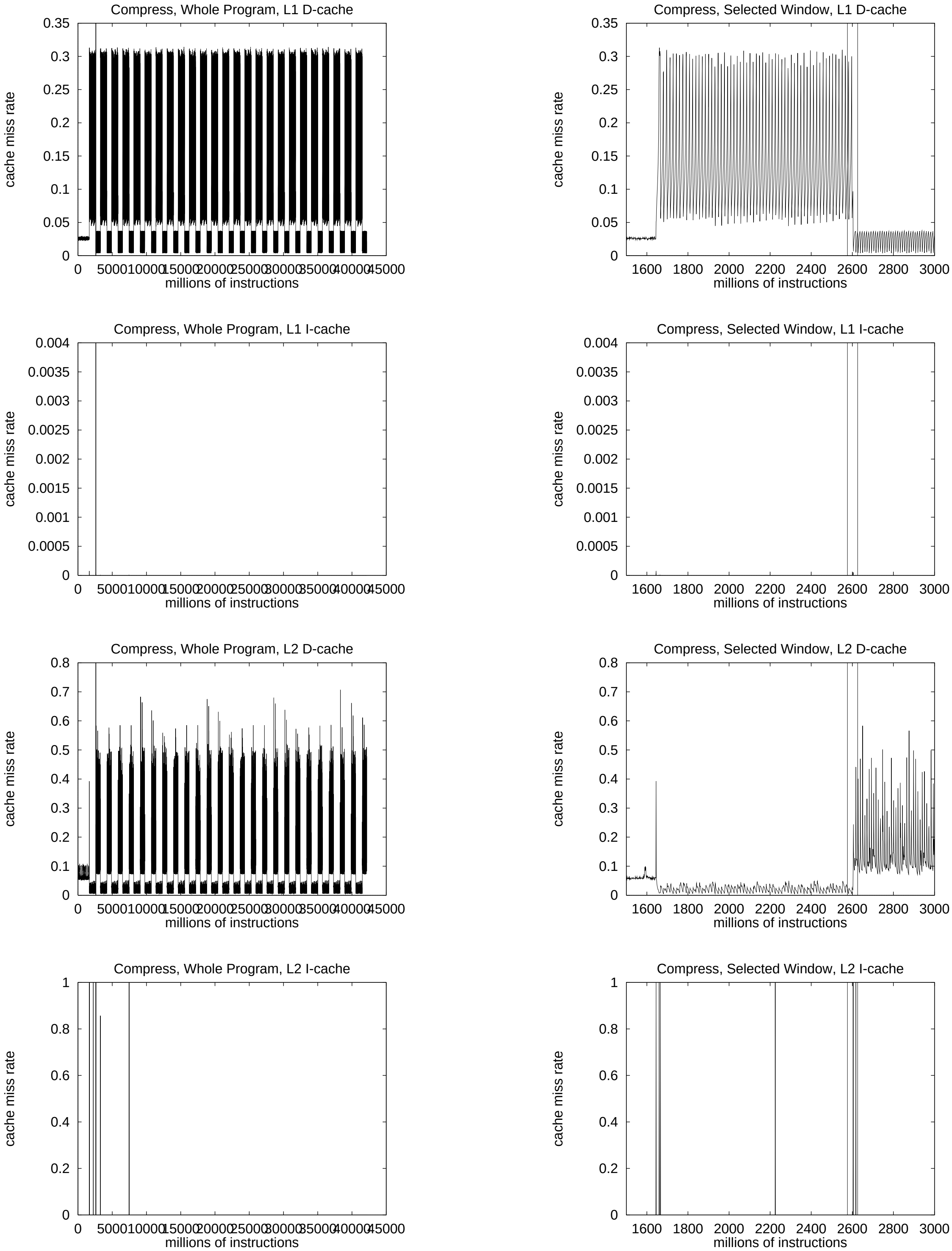

Figure 7: Interval cache miss rates for compress.

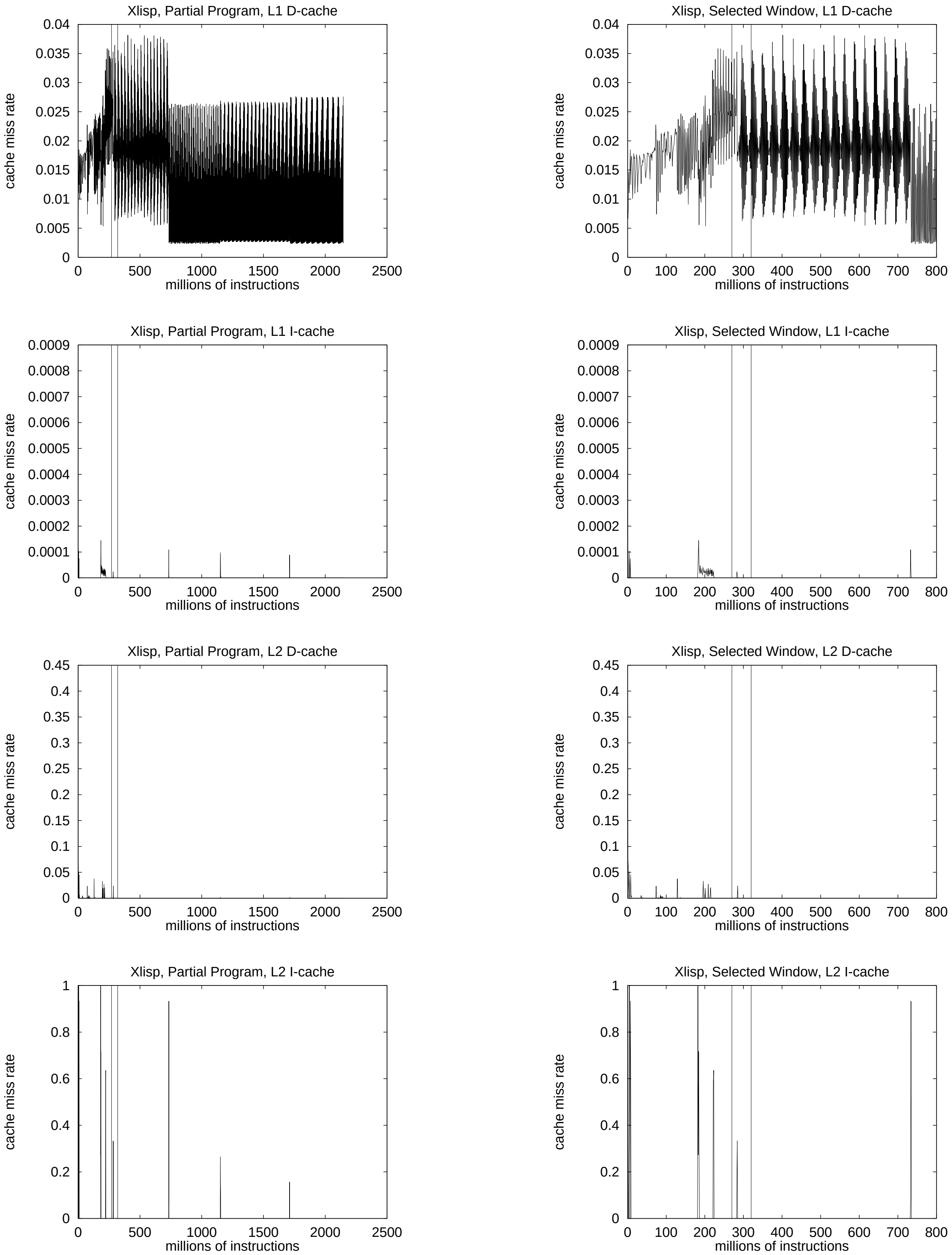

Figure 8: Interval cache miss rates for xlisp. Xlisp with all its inputs runs for about 55 billion instructions, but here we only show about 2 billion instructions.

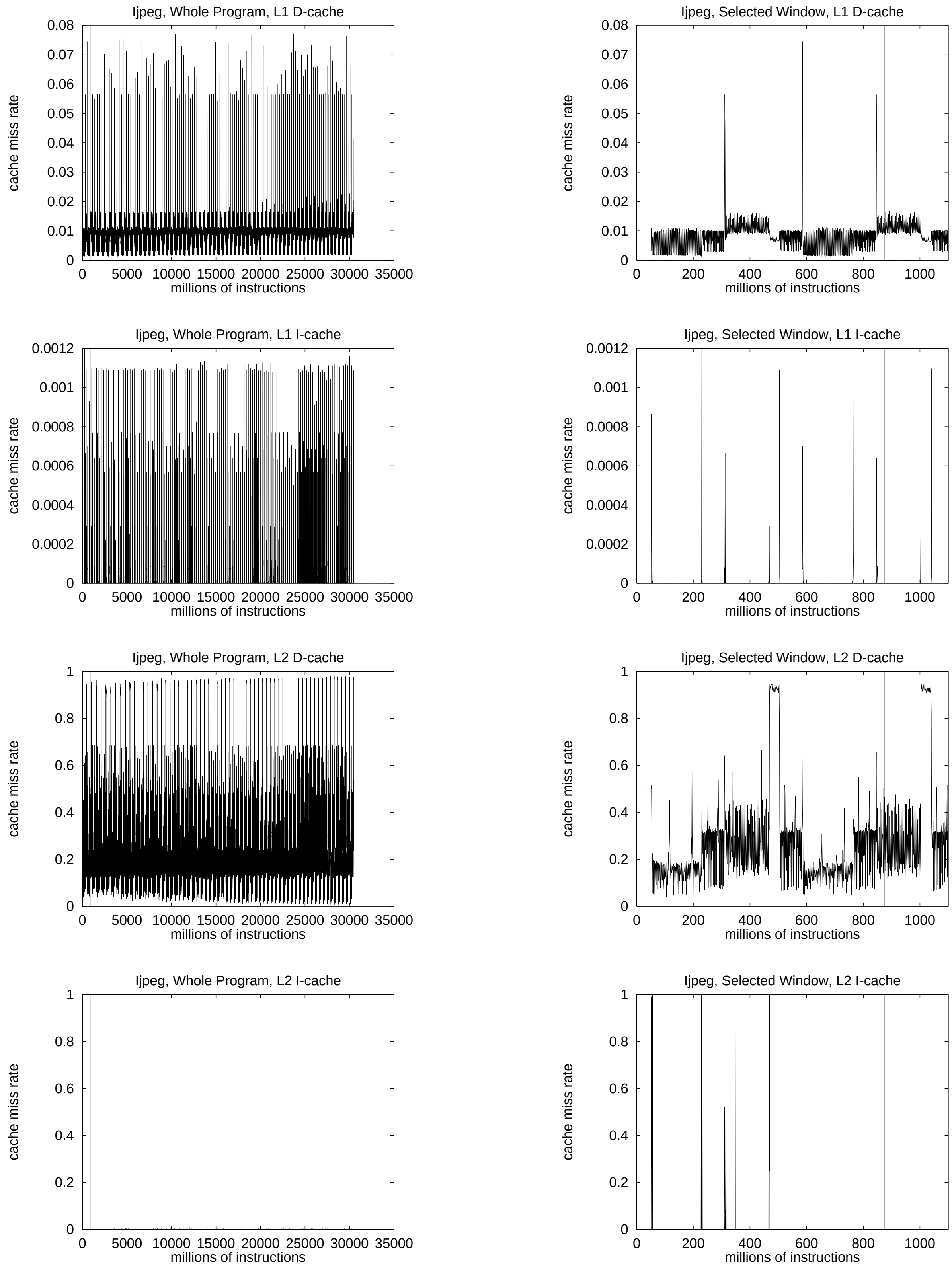

Figure 9: Interval cache miss rates for ijpeg.

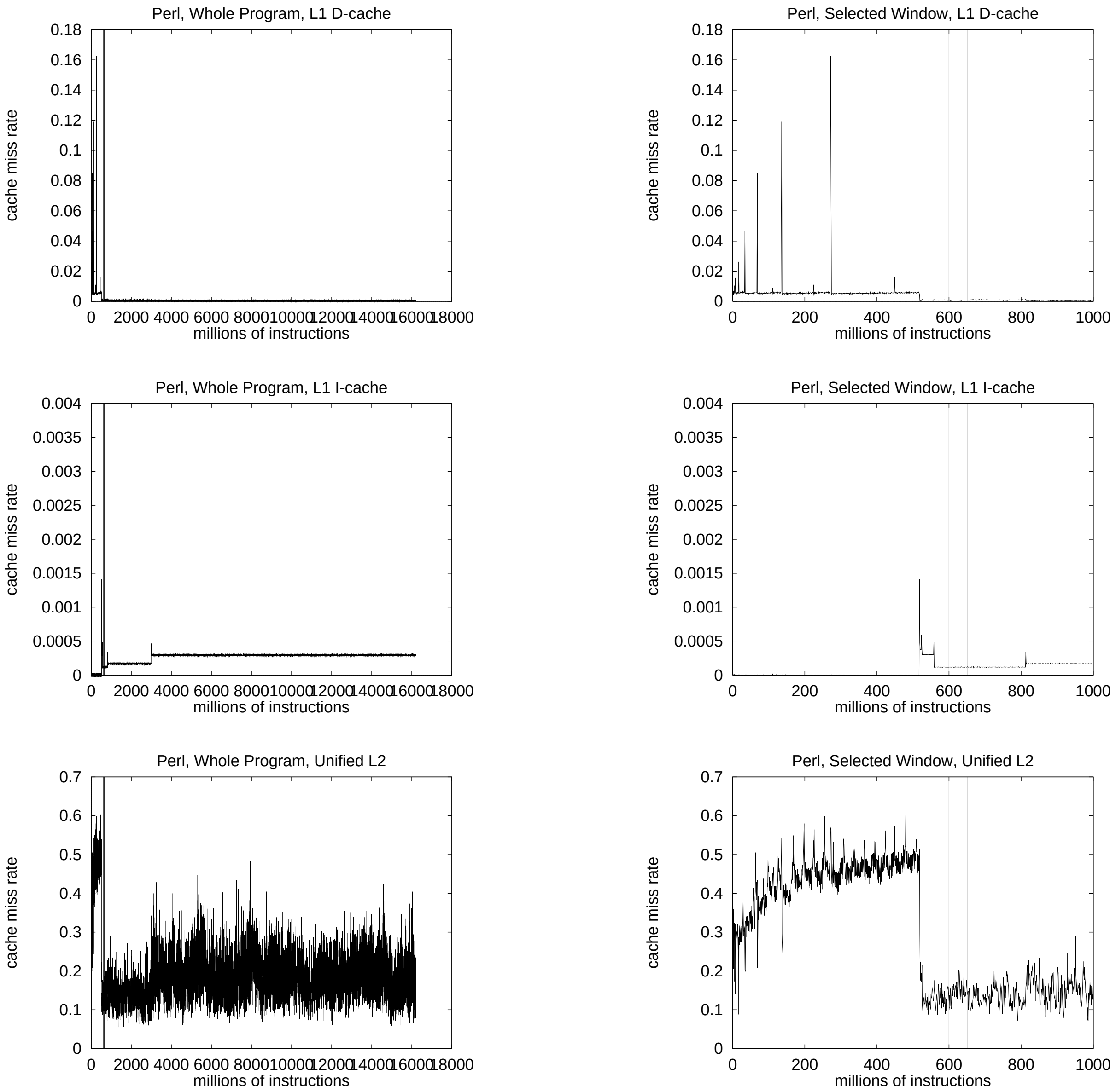

Figure 10: Interval cache miss rates for perl. Unified L2.

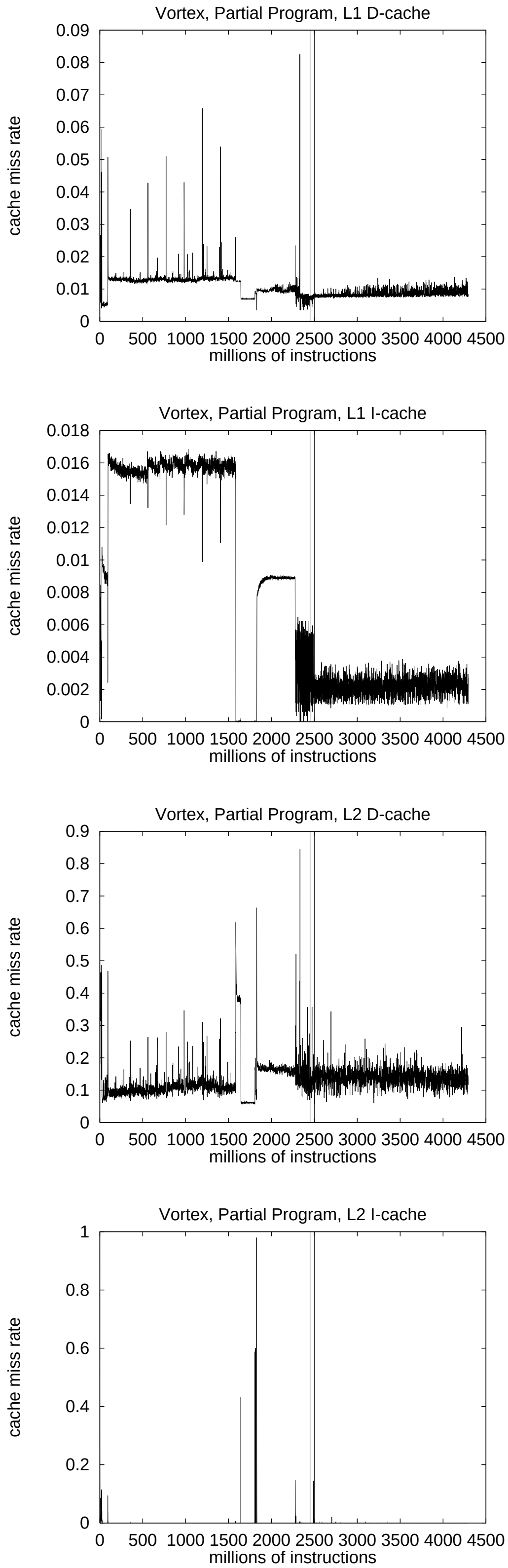

Figure 11: Interval cache miss rates for vortex. Vortex runs for nearly 80 billion instructions; here we only show the first 4 billion. Because this trace is so short compared to those in Figure 3, we do not show a second graph for each cache trace.

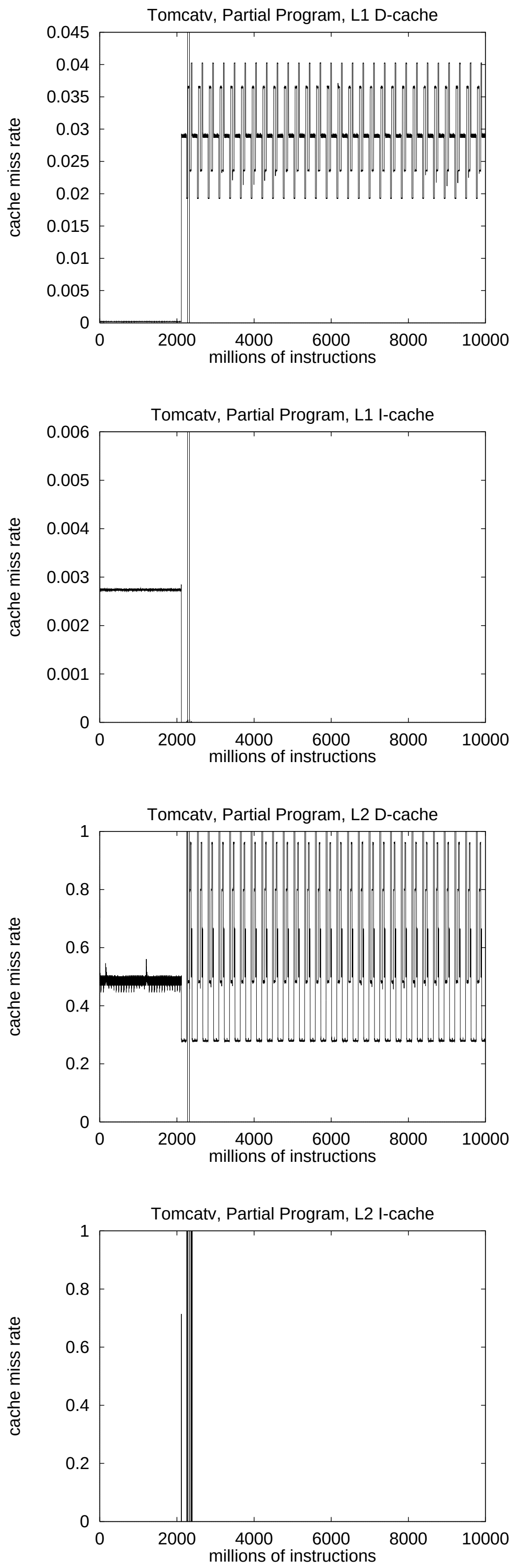

Figure 12: Interval cache miss rates for tomcatv. Because tomcatv with reference data runs for so long (210 billion instructions), a chart for the entire program shows no useful detail. Instead, we just plot the first 10 billion instructions.

5 Verifying the Simulation Windows

This section uses HydraScalar to perform cycle-level simulations and evaluate the representativeness of our chosen 50 M instruction simulation windows. We first show how detrimental unprepresentative startup behavior can be. Next, we compare the results of a carefully chosen 50 M instruction segment against the results obtained with a much longer simulation window. If the two simulation lengths show no significant difference, then there is no advantage to the longer simulation and this implies that the 50 M window is representative.

This evaluation tests not only the overall IPC, but also the relationship between cache size and RUU size. Each comparison reports the IPCs for a range of cache and RUU sizes. The resulting IPC surfaces are then compared.

5.1 The Impact of Programs' Startup Behavior

As mentioned with regard to *compress*, most programs show some sort of startup behavior. In several cases the corresponding misprediction rate 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. In particular, many research studies simulate onlyl the first 50–100 M instructions from each benchmark, and consequently include only startup behavior.

To show how significantly results can change when the 50 M instruction simulation window is dominated by startup behavior, Figure 13 plots three-dimensional graphs of IPC vs. RUU size vs. data cache size for *go, gcc, compress*, and *perl*, for both regular and 100% branch prediction. The left two columns' worth of graphs show data for baseline branch prediction, and the right two columns show data for 100%-direction prediction. In each case, an accurate 50 M instruction simulation window that avoids startup behavior (using the warmup times given in Table 4) is compared with a window taken from the beginning of the program's execution (using a warmup period of just one million instructions to prime processor structures). Within each graph, the vertical, or z-axis presents performance in terms of average instructions per cycle, or IPC. Each bar represents a single simulation with one combination of L1 cache 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.

For all benchmarks, 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 instructions of warmup eliminate the most egregious startup effects. For *go, compress*, 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 startup behavior have less impact on overall results. On the other hand, the startup phase can be extremely 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.

The simulation window should therefore be placed after any such initial phases. Fortunately, this can be done fairly quickly using fast-mode simulation. SimpleScalar has recently begun offering a checkpointing facility that removes even the need for fast-mode simulation [1]. Unless the initial behavior dominates the rest of the program's execution, omitting it still gives reasonable results.

5.2 Evaluating the Effectiveness of Small Simulation Windows

A 50 M instruction simulation window reliably gives representative results for SPECint programs, if the window is carefully chosen and is taken from later in the program's execution. As evidence for this, Figures 14 and 15 give further 3D graphs comparing the chosen 50 M window to a 250 M instruction simulation window (We have 500 M-instruction data for *go*, and 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. The one slight exception is *gcc*, for which the step from an 8 K to a 16 K cache is not as pronounced for 50 M as for 250 M. Data with a 100 M window does match 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

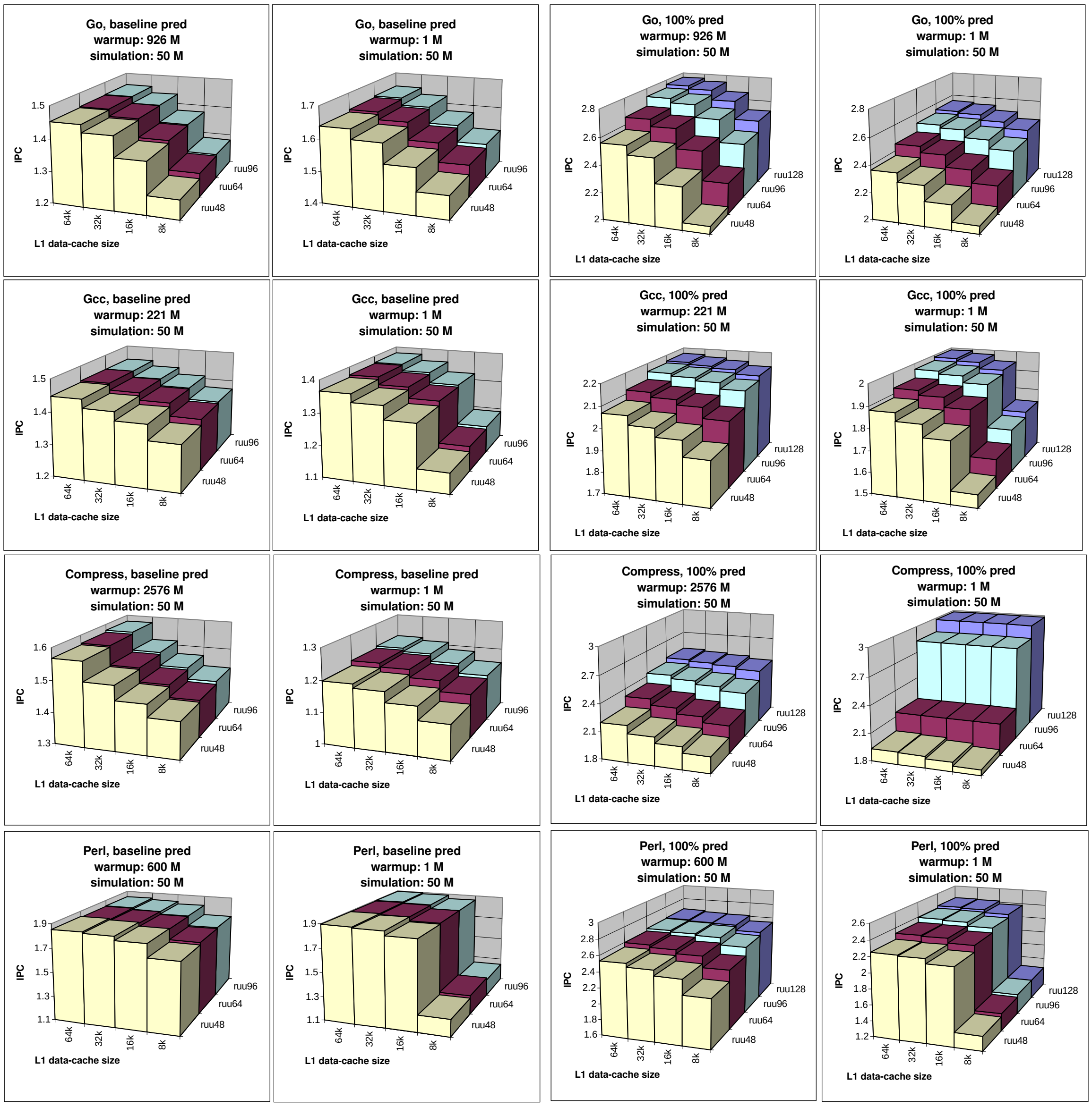

Figure 13: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations and no simulator warmup vs the chosen warmup interval.

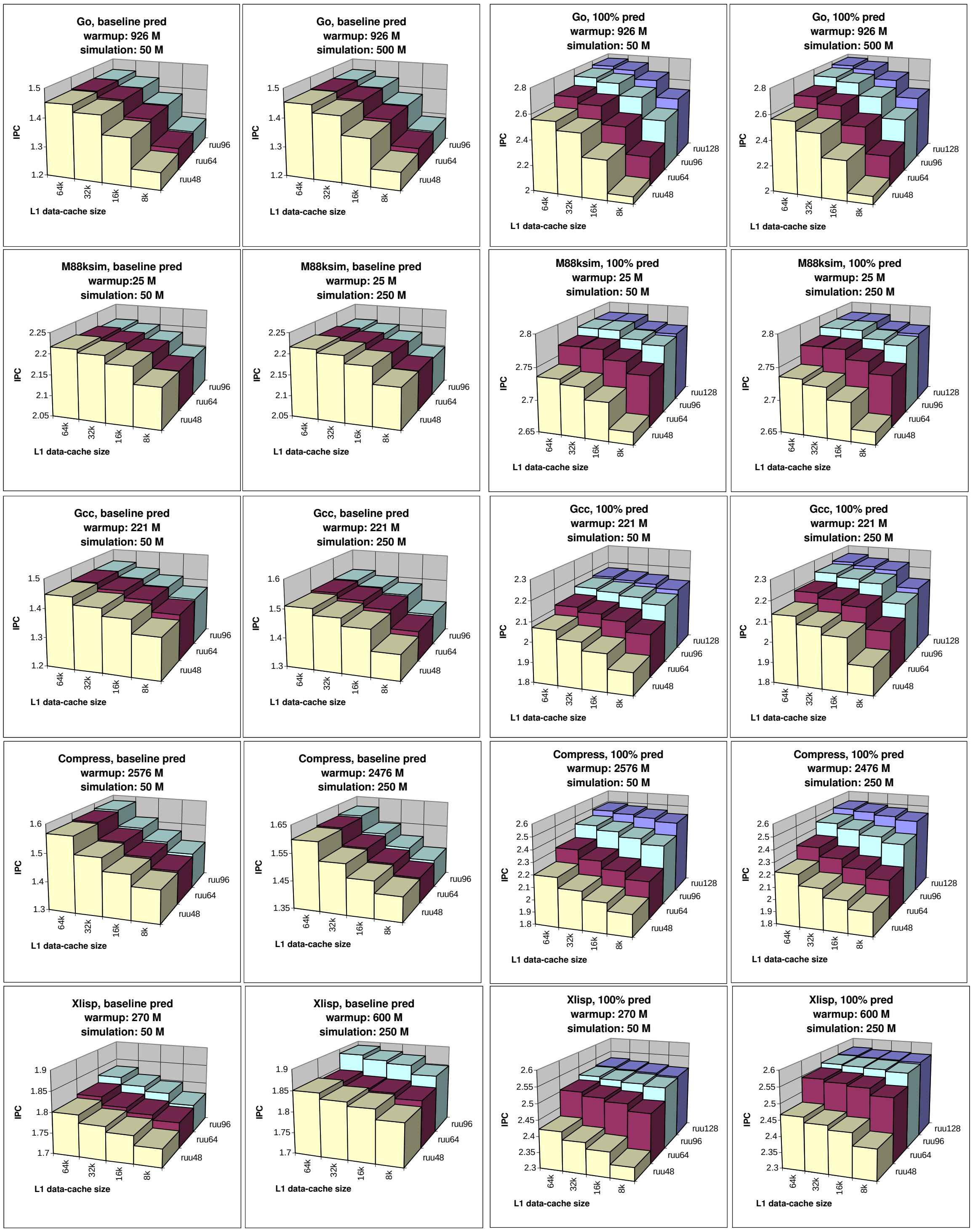

Figure 14: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations vs. 250 M-instruction simulations.

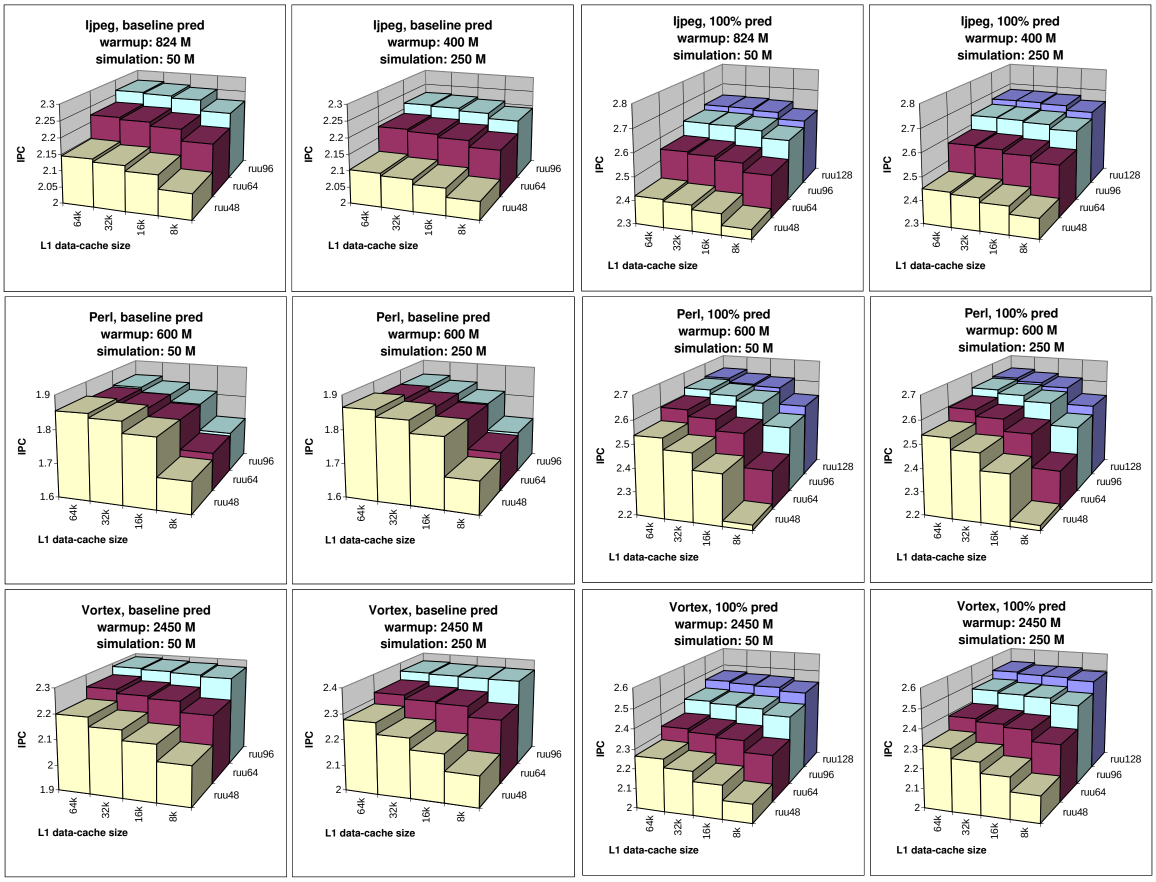

Figure 15: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations vs. 250 M-instruction simulations.

also sensitive to L2 misses, because the L2 miss behavior does not settle down until several execution phases have passed. But consulting the L2 interval-miss-rate plot when choosing a simulation window produced the 50 M instruction segment used here, and it does capture representative behavior.

Gathering the data for Figures 14 and 15 forced us to refine our initial choice of window in the case of *ijpeg* and *gcc* (by consulting the L2 trace). Choosing a representative 50 M-window window using the interval traces is a good heuristic, but verifying it by comparing 50 M-instruction plots of IPC vs. RUU size vs. D-cache size against 250 M-instruction plots provides even more reliable simulation windows.

Note that for programs with distinct phases of execution, like *compress* or *ijpeg*, the window has been placed so that it straddles the two phases (fortunately, these SPECint programs have at most two phases). Space considerations prevent us from presenting further 3D graphs, but 50 M-samples taken from just a single phase do not match behavior from the other phase. Instead, as Figures 14 and 15 show, a straddling 50 M-window captures behavior that corresponds well to a much larger 250 M straddling window that includes a large portion of each phase. The small straddling window therefore seems able to provide representative behavior even for programs like these that have two phases. Recall, however, that the phases in *compress* are an artifact of the SPEC benchmark version of this program. The different phases correspond to alternating passes of compressing and decompressing the same data. A better solution for *compress* might be to treat the two phases as two separate programs, with one simulation window in the middle of the compression phase, and one window in the middle of the decompression phase.

6 Conclusions and Future Work

In summary, this section has described a technique for choosing a short simulation window that is representative of the program's overall execution behavior. Of course, this technique can be used to select simulation windows of any length. But cycle-level simulations of quite long simulation windows have shown that the short, 50 million instruction windows provide accurate results. These short windows therefore avoiding long simulation times, while still permitting simulations with large, realistic inputs.

The steps involved in choosing a simulation window are:

1. Collect interval traces of branch misprediction rates and cache miss rates, including L2 miss rates. This can be done with instruction-level simulation.

2. Identify a candidate simulation window of the desired length (*e.g.*, 50 million instructions). A good candidate should

    - Avoid program startup behavior
    - Capture representative branch prediction and cache behavior.
    - Capture representative behavior for any other aspect of importance. For example, when studying the design of the floating-point unit, the distribution of floating-point operations or even their operands may need to be examined.

3. Compare the IPC-RUU-data cache size surfaces for the candidate simulation window and for a much longer simulation window (5–10 times as long). Ideally, the longer simulation window would consist of the program's entire execution, but for many programs this is infeasible even if only done once. Note that cycle-level simulations are required to produce these 3D graphs.

    When studying other aspects of processor design, like the floating-point unit, additional 3D graphs that test sensitivity to the feature in question should be generated. Satisfactory IPC-RUU-data cache surfaces are still necessary, but when exploring other design issues, not sufficient.

4. If the 3D surfaces match reasonably, a suitable simulation window has been found. If the surfaces do not match, return to step 2.

The drawback to this approach is that it must be done by hand, and is labor-intensive. In practice, however, the necessary experiments are simple, and they only need to be done once for each program to be added to the benchmark set. The advantage to this approach is that it allows short simulation times. The IPC-RUU-cache size surfaces are sufficiently sensitive to provide reasonable confidence in the quality of the chosen simulation window. As a result, this simulation methodology avoids the unrepresentative program behavior that plagues many simulation-based studies, while at the same time avoiding long simulation times.

There is a great deal of room for future work. We have only gathered interval traces for one branch-predictor and one cache configuration; it would be interesting to explore the impact on the traces of varying the configurations. We have also only used one interval—one million instructions—and it would be helpful to explore the impact of different interval sizes. We have only examined the "ref" inputs, and it would be useful to compare the differences between the "ref" and the shorter training or test inputs; as well as the differences among multiple "ref" inputs. Finally, we have only presented data for the SPECint95 suite and one SPECfp95 program; it would be very useful to collect interval traces for other common benchmarks, like the SPLASH-2 [25] suite, the IBS [24] suite, and new object-oriented and multimedia benchmarks being proposed.

References

[1] T. Austin. Personal communication, Mar. 1999.

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

[3] 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.

[4] 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.

[5] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 12–21, May 1996.

[6] 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*, pages 228–32, Nov. 1994.

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

[8] 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 of Parallel Programming*, 25(5):363–83, Oct. 1997.

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

[10] 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*, pages 90–95, Oct. 1998.

[11] 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.

[12] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11):1325–1336, Nov. 1988.

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

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

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

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

[17] K. Skadron. *Characterizing and Removing Branch Mispredictions.* PhD thesis, Princeton Univ., June 1999.

[18] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers.* To appear.

[19] 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.

[20] 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.

[21] K. Skadron, D. W. Clark, and M. Martonosi. Speculative updates of local and global branch history: A quantitative analysis. Tech. Report TR-589-98, Princeton University Dept. of Computer Science, Dec. 1998.

[22] 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.

[23] The Standard Performance Evaluation Corporation. WWW Site. http://www.specbench.org, Dec. 1996.

[24] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 345–56, June 1995.

[25] 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.

[26] 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.