IPC Considered Harmful for Multiprocessor Workloads

MANY ARCHITECTURAL SIMULATION STUDIES USE INSTRUCTIONS PER CYCLE (IPC) TO ANALYZE PERFORMANCE. FOR MULTITHREADED PROGRAMS RUNNING ON MULTIPROCESSOR SYSTEMS, HOWEVER, IPC OFTEN INACCURATELY REFLECTS PERFORMANCE AND LEADS TO INCORRECT OR MISLEADING CONCLUSIONS. WORK-RELATED METRICS, SUCH AS TIME PER TRANSACTION, ARE THE MOST ACCURATE AND RELIABLE WAY TO ESTIMATE MULTIPROCESSOR WORKLOAD PERFORMANCE.

Alaa R. Alameldeen
Intel Corporation

David A. Wood
University of Wisconsin-Madison

Computer architects use simulation as a primary tool to evaluate computer system performance and to compare architectural alternatives. Simulation studies frequently compare the performance of one or more architectural enhancements against a base system. For many such studies, instructions per cycle (IPC) is the performance metric of choice. An increase in IPC for an architectural enhancement represents a performance benefit over the base system.

In this article, we challenge the commonly held view that IPC accurately reflects performance—at least for multithreaded workloads running on multiprocessors. Our simple counterexamples show cases in which IPC increases do not reflect a performance gain, and others in which IPC decreases do not reflect a performance loss. In some of our examples, IPC actually decreases as performance increases, and vice versa. As the number of processors increases, IPC becomes a less accurate measure of performance.

The IPC measurement's inaccuracy stems from the incorrect assumption that *instructions per program* remains constant across all executions. In fact, the instruction path of multithreaded workloads running on multiple processors can vary substantially.[1] Spin locks and other synchronization mechanisms magnify small timing variations into very different execution paths. This is especially true for commercial workloads that spend significant time in the operating system, where contention for system resources can result in different scheduling decisions and increased idle time.

Several researchers have used various approaches to improve IPC's accuracy—ignoring system code, excluding lock overhead and idle time, or using trace-driven simulation, for example. These approaches, however, have their own drawbacks. Work-related metrics present a safer, more accurate alternative because they are directly proportional to the gold-standard performance measure, *time per program*.

Performance evaluation using IPC

Many computer architecture textbooks and introductory courses teach that the time to run an application (time per program) is the ultimate performance measure for an archi-

Published by the IEEE Computer Society

tecture.[2] Other metrics can provide insight, but only to the extent that they accurately reflect time per program. To compare the performance of a program P on a new system A and a baseline system B, the speedup of A over B is the ratio of P's runtime on B divided by its runtime on A:

$$\text{speedup(A)} = \frac{(\text{time/program})_B}{(\text{time/program})_A}$$

Because runtime gives little insight into microarchitectural behavior, architects have long used the Iron Law to deconstruct program performance:[2]

$$\text{time/program} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$

When substituting terms in this equation, most performance studies assume that the first and last terms (instructions per program and time per cycle) are the same for both systems A and B. This means that the speedup is a function of the second term only—cycles per instruction (CPI), or its inverse, instructions per cycle (IPC):

$$\text{speedup(A)} = \frac{\text{CPI}_B}{\text{CPI}_A} = \frac{\text{IPC}_A}{\text{IPC}_B}$$

This simplified form is accurate only if the assumptions hold: that is, if both systems A and B have the same cycle time (a fixed time per cycle) and also run the same number of instructions (the same instructions per program). The latter assumption essentially equates the number of instructions executed with the amount of useful work performed by the application. This assumption generally holds for the complete execution of single-threaded, uniprocessor, user-level programs. However, programs exhibit phase behavior, where IPC can differ vastly between one phase and another. Researchers have proposed several sampling techniques to accurately estimate IPC in uniprocessor programs that exhibit phase behavior.[3]

Multithreaded programs present a more fundamental problem. Previous work has shown that small timing variations, such as the exact interrupt timing or races for a mutex (mutual exclusion) lock, can cause the operating system to make different scheduling decisions.[1] These divergent paths result in different combinations of thread phases, in which a program might execute a substantially larger or smaller number of instructions to perform the same amount of useful work.

Architectural enhancements can, and do, introduce timing variations that lead to divergent execution paths. For example, a large cache or a better prefetching algorithm eliminates misses that can favor one thread over another. If this timing difference changes which thread reaches a mutex lock first, the enhanced system might execute a very different path than the base system. System-level behavior accentuates these effects—for example, by making different scheduling decisions. Seemingly small architectural changes can significantly affect how much time a thread spends executing idle-loop instructions, spin-lock wait instructions, or various system-level privileged code instructions, such as a translation look-aside buffer (TLB) miss handler. Although such instructions significantly change IPC, they have little effect on the amount of useful work a user program actually accomplishes.

When IPC is misleading

We've performed several architectural simulation studies to demonstrate that using IPC can lead to incorrect conclusions. As our baseline configuration, we used an eight-processor chip multiprocessor (CMP) system with a 5-GHz clock and out-of-order Sparc V9 processors. We used the Simics full-system simulator,[4] extended with GEMS (General Execution-Driven Multiprocessor Simulator, a detailed memory-system and out-of-order processor timing simulator).[5] We modeled our base system as a future-generation CMP inspired by IBM's Power5[6] and Sun's Niagara,[7] although our chip has only single-threaded cores. The base system has the following specifications:

- Each processor has private split L1 instruction and data caches. Each instruction or data cache is 64 Kbytes, is four-way set-associative, and has a three-cycle access time.

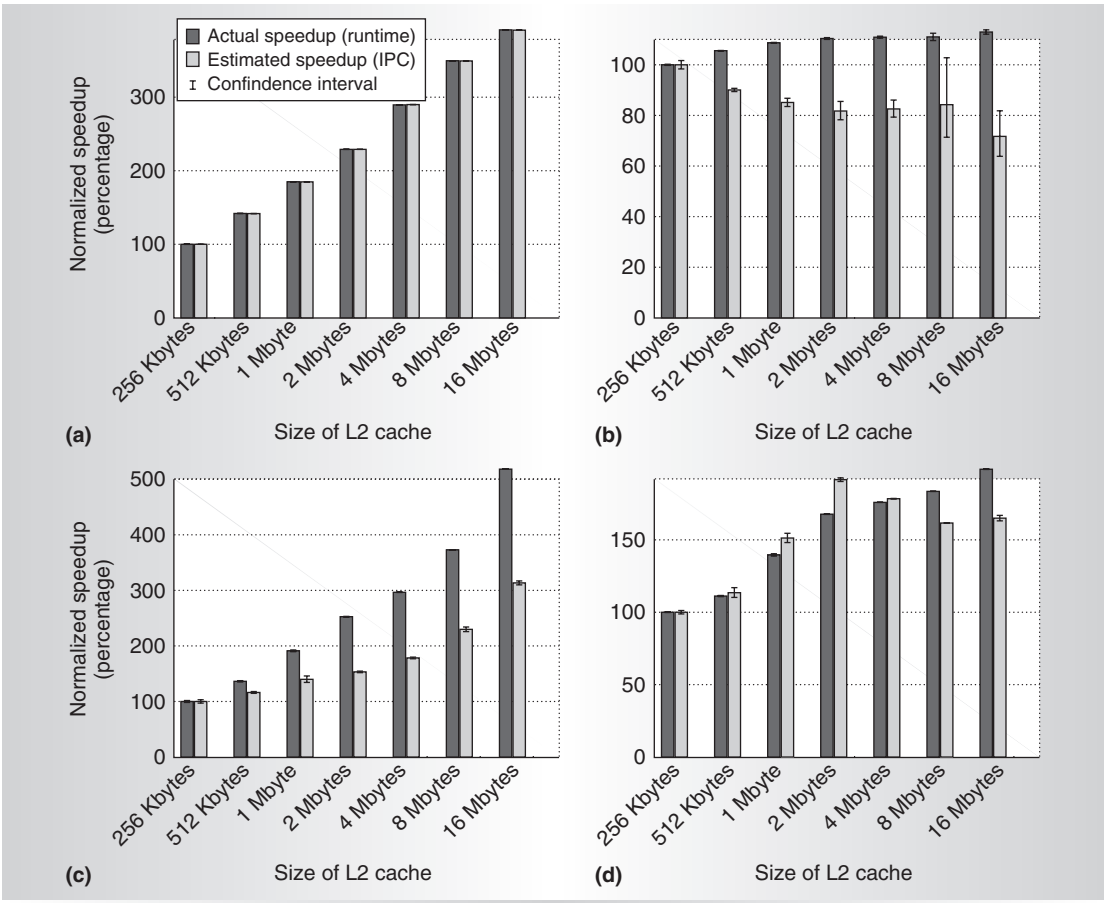

Figure 1. Normalized speedup, calculated using runtime and IPC, over a 256-Kbyte L2 cache baseline: jbb (a), fma3d (b), apache (c), and mgrid (d).

- The unified, shared 4-Mbyte L2 cache (unless otherwise specified) is composed of eight 512-Kbyte banks. The L2 cache is eight-way set-associative and has 64-byte lines and a 15-cycle L2 hit latency.
- 400-cycle memory latency.
- 20-Gbytes/s aggregate pin bandwidth.

We used four multithreaded commercial workloads from the Wisconsin Commercial Workload Suite, and four benchmarks from the SPEComp2001 suite, all running under the Solaris 9 operating system. (See the "Workload descriptions" sidebar for details.) For each data point in our results, we present the average and the 95 percent confidence interval of multiple simulations to account for space variability.[1] Our speedup results for the commercial workloads represent the average speedup in number of cycles per transaction (or request). For the SPEComp benchmarks, our speedup results represent the average speedup in number of cycles required to complete the main loop. We compared these speedups to speedups calculated using IPC.

Experiment 1: L2 cache size

We performed a simple experiment in which we varied the L2 cache size between 256 Kbytes and 16 Mbytes (all with a 15-cycle hit latency), and compared the speedups over the 256-Kbyte base case. Figure 1 shows the runtime and IPC speedup we calculated using data from the same runs, for four representative benchmarks. (We show only four benchmarks because zeus, oltp, and apsi behave similarly to apache, and art behaves similarly to jbb.)

The speedup computed using runtime represents the gold standard with which to compare other measurements since it simply normalizes time per program. Speedup computed using IPC also accurately pre-

dicts performance when instructions/program remains roughly constant. Figure 1a shows that for jbb, IPC speedup matches runtime speedup almost exactly. For workloads similar to jbb, using IPC as a performance estimate is justified. Unfortunately, using IPC for other workloads leads to conclusions that are incorrect—either in direction or magnitude.

IPC leads to wrong conclusions. Figure 1b shows the runtime and IPC speedup for the fma3d benchmark. By the runtime measurement, performance improves slowly as cache size increases; for example, a 16-Mbyte L2 cache improves performance by 13 percent. Conversely, the IPC results suggest that increasing cache size actually degrades performance; by this measurement, a 16-Mbyte L2 cache degrades performance by 28 percent. Clearly, using IPC leads to the erroneous conclusion that increasing the cache size hurts performance. Moreover, the variability in IPC-based results (demonstrated by the confidence interval error bars) is much larger than the variability in runtime results for the same set of runs.

Here, it is obvious that the conclusion drawn from the IPC speedup is wrong. In more subtle cases, when the architectural enhancement's effect on performance is not already known, a false conclusion based on IPC could be harder to detect.

IPC underestimates performance. Figure 1c shows the runtime and IPC speedups for the apache benchmark, whose behavior is also representative of that of zeus, oltp, and apsi (not shown). As this graph shows, increasing the L2 cache size leads to a much greater speedup in runtime than in IPC. Using IPC as a performance metric underestimates the performance improvement by nearly a factor of 2. For example, increasing the L2 cache size from 256 Kbytes to 16 Mbytes results in a runtime more than 5 times as fast, but improves the IPC by only about a factor of 3.

IPC overestimates performance. Figure 1d shows the runtime and IPC speedup for the mgrid benchmark. In many configurations, using IPC results in a greater speedup than using runtime. For example, using IPC shows a 92

Workload descriptions

We used four multithreaded commercial workloads from the Wisconsin Commercial Workload Suite,[1] and four benchmarks from the SPEComp2001 suite.[2]

Online transaction processing (OLTP)

Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 5-Gbyte database with 25,000 scaled-down warehouses on eight raw disks and a single log disk. We simulate 16 users per processor, warm up the database for 100,000 transactions, and measure for 100 transactions.

Java server workload: SPECjbb

We use Sun's HotSpot 1.4.0 Server JVM, 1.5 threads and 1.5 warehouses per processor (12 for eight processors), a data size of approximately 44 Mbytes, a warm-up interval of 200,000 transactions, and measurement intervals of 2,000 transactions.

Static Web serving: apache

We use apache 2.0.43 for Sparc and Solaris 9, configured to use pthread locks and minimal logging at the Web server. We use Surge[3] to generate Web requests, a repository of 20,000 files (totaling about 500 Mbytes), and we disable apache logging for high performance. We simulate 400 clients per processor, each with 25 ms between requests; warm up for about 2 million requests; and measure for 500 requests.

Static Web serving: zeus

Zeus uses an event-driving server model driven by Surge, where each processor is bound by a zeus process, which waits for Web-serving events (for example, open socket, read file, send file, and close socket). Its remaining configuration parameters are the same as those for apache.

SPEComp

We used four benchmarks from the SPEComp2001 benchmark suite:[2] 330.art, 324.apsi, 328.fma3d, and 314.mgrid. We used the ref input set, fast-forwarded each benchmark to the beginning of the main loop, warmed up caches for about 2 billion instructions, and measured until the end of the loop iteration.[4]

References

1. A.R. Alameldeen et al., "Simulating a $2M Commercial Server on a $2K PC," *Computer,* vol. 36, no. 2, Feb. 2003, pp. 50-57.
2. V. Aslot et al., "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," *Proc. Workshop on OpenMP Applications and Tools* (WOMPAT 2001), Lecture Notes in Computer Science 2104, Springer-Verlag, 2001, pp. 1-10.
3. P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems* (Sigmetrics 98 and Performance 98), ACM Press, 1993, pp. 151-160.
4. B.M. Beckmann and D.A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," *Proc. 37th IEEE/ACM Int'l Symp. Microarchitecture* (Micro 37), IEEE CS Press, 2004, pp 319-330.

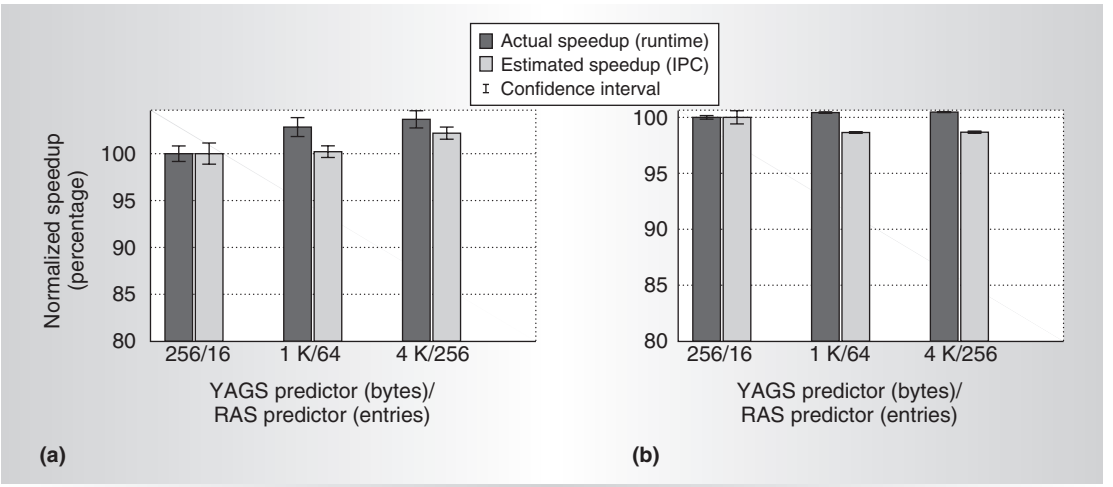

Figure 2. Normalized runtime and IPC speedup over the baseline of a 256-byte YAGS direct branch predictor and a 16-entry return address stack (RAS) predictor: apache (a) and mgrid (b).

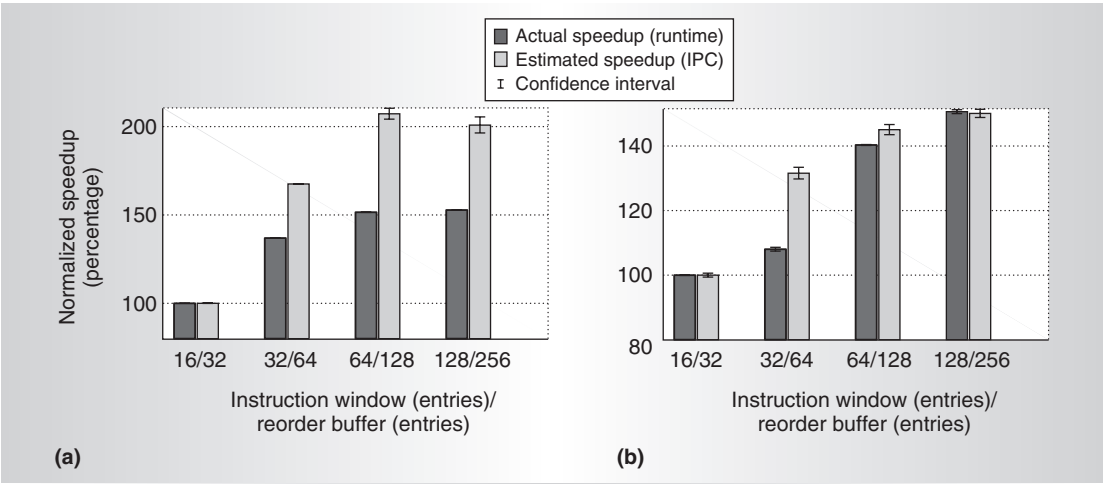

Figure 3. Normalized runtime and IPC speedup over a 16-entry instruction window, 32-entry reorder buffer baseline: fma3d (a) and apsi (b).

percent speedup for the 2-Mbyte cache over the baseline 256-Kbyte cache, while using runtime shows only a 68 percent speedup.

Experiment 2: Branch prediction

Figure 2 shows the speedup of runtime and IPC for two branch-predictor alternatives over a baseline of a 256-byte YAGS ("yet another global scheme") direct branch predictor[8] and a 16-entry return-address stack predictor.[9] The predictors' size has a limited effect on our benchmarks. IPC underestimated speedup for apache and mgrid, even suggesting the incorrect conclusion that a bigger branch predictor degrades mgrid's performance.

Experiment 3: Reorder buffer size

Figure 3 shows the impact of instruction window and reorder buffer scaling on two representative benchmarks. Although scaling greatly improves performance for both benchmarks, IPC significantly overestimates speedup. For example, for the 64/128 system (64 instruction window entries and 128 reorder buffer entries), fma3d's runtime speedup was approximately 50 percent, whereas the IPC more than doubled.

Experiment 4: More threads

Figure 4 illustrates that more threads generally cause more IPC error. In this experi-

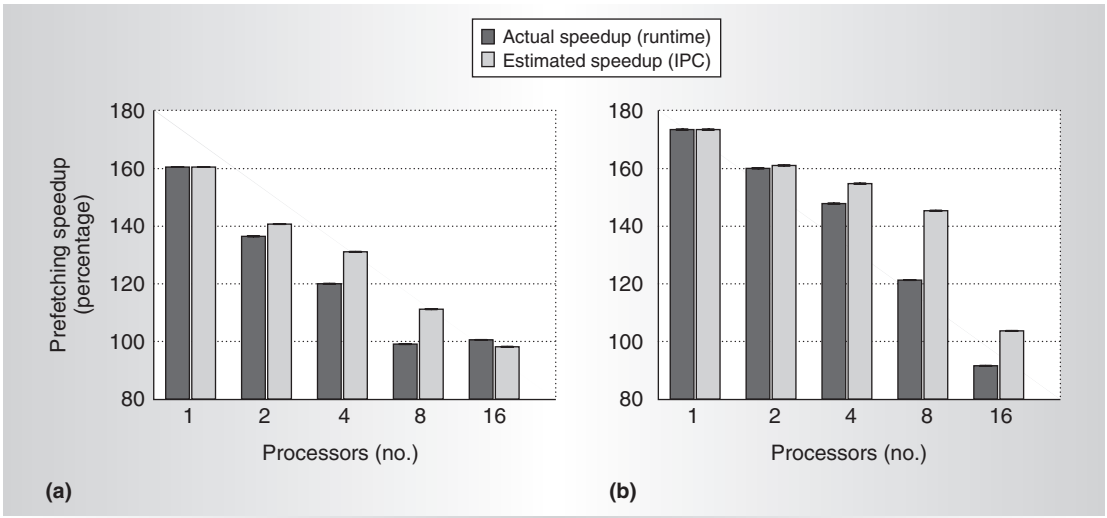

Figure 4. Speedup for runtime and IPC over a baseline without prefetching: apache (a) and zeus (b). Speedup for each configuration is relative to the baseline for the same processor configuration—not to a single processor.

ment, we show the speedup of a hardware stride-based prefetcher—similar to that implemented in the Power5[6]—for different processor configurations compared to a baseline with the same number of processors but no prefetching. (Speedup is shown relative to each processor configuration, not to be confused with parallel speedup over a uniprocessor.) Figure 4 shows that IPC overestimates speedup in most cases. In addition, the relative error in IPC speedup generally increases as the number of processors increase. In some cases—such as zeus on 16 processors—this error can lead to the opposite of the correct conclusion.

Incomplete solutions to IPC problems

IPC speedup differs from runtime speedup when an architectural enhancement changes the program's execution path and this new path performs more or less useful work per instruction. For example, instructions executed in the idle loop, spin-lock loops, and some system calls do not directly contribute to a program getting work done. When an architectural enhancement disproportionately decreases the number of such instructions, IPC can decrease even if runtime improves. Conversely, when an architectural enhancement increases the number of idle or spin-lock instructions, IPC can be inflated with little effect on runtime.

Previous studies have used several techniques to mitigate the problems with IPC. However, these techniques are either infeasible for commercial workloads or do not sufficiently address the problem.

Using user-code IPC

Many of the drawbacks of using IPC arise from the simulation of system code, which does not correspond directly to work done. Many studies—including those based on SimpleScalar[10]—have used the simple solution of estimating IPC for user-level code and not for system code. For many commercial benchmarks, however, this approach is infeasible, because system code constitutes a significant fraction of their runtime.

Table 1 shows the percentages of dynamic and static instructions and cache misses (L1 instruction, L1 data, and L2) due to system code for our eight benchmarks. System code has a significant impact on all the benchmarks except jbb. For apache, zeus, and art, the majority of all dynamic and static instructions execute in kernel mode. System code also constitutes nearly a quarter of oltp's dynamic instructions. Even for apsi and mgrid, where the percentage of dynamic system code instructions is insignificant, the majority of all static instructions belong to the operating system, which results in a significant fraction of all L1 instruction cache misses.

These results clearly demonstrate that ignoring system code when evaluating per-

Table 1. Impact of system code.

	Instructions due to system code (percent)		Cache misses due to system code (percent)		
Benchmark	**Dynamic**	**Static**	**L1I**	**L1D**	**L2**
apache	86.6	61.1	69.5	91.4	85.5
zeus	80.0	80.1	66.8	80.9	72.2
oltp	23.4	21.7	23.2	44.7	33.0
jbb	2.2	20.3	3.7	4.7	3.2
apsi	0.3	89.1	57.9	0.4	0.7
art	60.5	98.0	84.5	19.7	7.3
fma3d	4.2	81.2	53.1	2.0	2.1
mgrid	1.3	57.7	33.7	2.6	2.2

Table 2. Atomic instructions.

	Static atomic instructions		Dynamic atomic instructions	
Benchmark	**Total number**	**Number in user code**	**Total percentage**	**Percentage in user code**
apache	53	17	0.70	0.02
zeus	30	0	0.60	0.00
oltp	167	126	0.20	0.15
jbb	320	305	0.70	0.70
apsi	18	1	0.00	0.00
art	21	0	0.01	0.00
fma3d	18	2	0.06	0.00
mgrid	20	4	0.02	0.00

formance leads to inaccurate conclusions for most of our benchmarks. Furthermore, system code also accounts for a significant portion of the SPECcpu benchmark executions.

Ignoring idle time

Another source of error from using IPC is that idle loop instructions do not correspond to work done. But because running the idle loop is fast, including these instructions artificially inflates IPC without a similar improvement in runtime.[11] However, excluding idle loop instructions is neither easy nor sufficient.

In real operating systems, the idle loop corresponds to a large number of instructions in multiple functions. For example, the idle loop in Open Solaris is a part of the operating system dispatcher code and corresponds to hundreds of lines of C code if we include function calls (see http://cvs.opensolaris.org/source/xref/on/usr/src/uts/common/disp/disp.c).

Moreover, isolating the idle loop instructions is not sufficient to resolve the discrepancies in speedup between IPC and runtime, since most well-tuned applications are idle only for a tiny fraction of their runtime. In our experiments, the idle time represented less than 5 percent of all workloads' runtime for the baseline configurations.

Ignoring spin locks

Spin locks and atomic instructions can cause significant deviations when IPC is the performance estimate. Spin-lock loops can add a significant number of instructions to a program's execution, inflating IPC without improving runtime. When the number of threads or processors increases, such loops and atomic operations are more frequent, widening the difference between IPC and runtime. Excluding spin-lock loops and atomic instructions might be perceived as a way to bridge the gap, but isolating these operations is not trivial and is not likely to be effective.

Table 2 shows the total number of static atomic instructions in our programs and those in user code. The large numbers of static atomic operations in many benchmarks demonstrate that eliminating them is difficult, especially since significant proportions exist in user code.

Furthermore, removing such instructions can significantly affect a program's execution, and eliminates the benefit from many architectural enhancements that target atomic operations. Table 2 also shows the percentages of all user-code dynamic instructions that are atomic; these are below 1 percent for all benchmarks, indicating that eliminating the impact of atomic instructions on IPC is unlikely to bridge the performance gap between IPC and runtime.

Trace-driven simulation

The central problem with the IPC measurement is that different execution paths result in the execution of different instructions. Trace-driven simulation eliminates this problem, since the trace instructions executed on different systems would be exactly the same. Lepak, Cain, and Lipasti used a methodology to obtain a similar effect using execution-driven simulation by eliminating nondeterminism.[12] However, neither approach is appropriate for evaluating architectural designs that affect thread interaction or the ordering of thread executions. Using traces or forcing determinism can lead to executions that would never occur for the architectural enhancement under study. Lepak, Cain, and Lipasti conclude that their deterministic simulation should be avoided when nondeterminism is high, and should be used with care in evaluating changed or relaxed architectural semantics.

Solution: Work-related metrics

So far, we have demonstrated the weaknesses of using IPC to evaluate performance for multithreaded workloads running on multiprocessors and have shown that previous proposals to address IPC's problems are incomplete or infeasible for commercial workloads. To address this situation, architects must return to first principles and use metrics proportional to time per program. Because it is not feasible to simulate the complete execution of many programs, we argue for simple work-related metrics—for example, time per work unit, or work units per time. These metrics will accurately predict performance if the unit of work is representative of the entire program.

Throughput-oriented applications

Work-related metrics are the norm for throughput-oriented applications such as our set of commercial workloads. For example, the TPC-C benchmark reports performance in transactions per minute for a measurement interval of at least 2 hours after the benchmark reaches a steady state (see http://www.tpc.org/tpcc). To limit simulation time for architectural studies, we recommend measuring the time required to complete a fixed number of transactions (or requests) after a suitable warm-up time to eliminate cold-start effects.

Using work-based metrics to estimate performance can lead to nondeterministic performance results. This behavior stems from small timing changes that lead different systems to execute different combinations of transactions. However, we can use statistical techniques to compensate for nondeterminism by using multiple runs to obtain tight confidence intervals.[1]

Scientific applications

Choosing an appropriate unit of work can be more difficult for scientific applications such as SPECcpu or SPEComp benchmarks. For simple iterative algorithms, in which a main loop dominates the computation time, a single iteration is the obvious work unit. More generally, we can define the unit of work to be the execution between a starting point and an end point in the program. An end point should be a place in the program where all executions can reach—for example, a fixed static instruction that occurs only once. For instructions that the program executes more than once, an end point can be a fixed dynamic instance of such an instruction or a future static instruction. For example, you can define an end point as the tenth dynamic instance of a static loop instruction or the first instruction outside the loop.

To estimate performance of scientific applications, simulations should measure the time required to finish a fixed unit (or units) of work. As long as the work unit is representative of the whole program, this approach is far more accurate than the commonly used IPC approach of counting the number of cycles required to finish a fixed number of instructions. The IPC approach is misleading, because it does not guarantee that the different systems will reach the same point in their execution, so the various systems could have finished a different amount of work.

Many IPC-based simulation studies use

sampling techniques—such as SimPoint[3]—to reduce simulation time or to obtain representative program fragments.[13] To use a work-related metric for such sampling techniques, we can define an end point for each sampling interval. This converts IPC-based performance estimates into work-based estimates, overcomes the shortcomings of using IPC, and makes speedup comparisons more viable. Luo and John further demonstrated that the error in runtime speedup is significantly lower than that from using CPI, and that accurately obtaining tight confidence intervals requires a shorter sampling period.[14]

For multithreaded, multiprocessor applications, IPC is a misleading performance measure that only becomes less accurate as the number of processors increases. The widespread shift to chip multiprocessors—also known as multicore processors—means that IPC will be a poor metric for many, if not most, future performance studies. As we've shown, the IPC-based alternatives also have significant drawbacks. Using work-related metrics, however, avoids these pitfalls by holding true to the time per program ideal. One challenge that remains is to find a robust method to deconstruct multithreaded program performance, in the same way that researchers have factored CPI (the inverse of IPC) to gain intuition into the impact of microarchitectural design decisions. MICRO

Acknowledgments

This work took place while Alaa R. Alameldeen was a graduate student at the University of Wisconsin-Madison. Partial support for this work came from the National Science Foundation, through grants CNS-0205286 and CCR-0324878, and from Intel and Sun Microsystems. (David A. Wood has a significant financial interest in Sun Microsystems.) The views expressed in this article are not necessarily those of the NSF, Intel, or Sun Microsystems. We thank Virtutech AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Mark Hill, the University of Wisconsin Computer Architecture Affiliates, the Wisconsin Multifacet project, Milo Martin, and our anonymous reviewers for their helpful feedback on this work.

References

1. A.R. Alameldeen and D.A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE Press, 2003, pp. 7-18.
2. D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* 3rd ed., Morgan Kaufmann, 2005.
3. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS-X), ACM Press, 2002, pp. 45-57.
4. P.S. Magnusson et al., "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, Feb. 2002, pp. 50-58.
5. M.M.K. Martin et al., "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *ACM Sigarch Computer Architecture News,* vol. 33, no. 4, Nov. 2005, pp. 92-99.
6. B. Sinharoy et al., "Power5 System Microarchitecture," *IBM J. Research and Development,* vol. 49, no. 4/5, 2005, pp. 505-522.
7. P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," vol. 25, no. 2, *IEEE Micro,* Mar.-Apr. 2005, pp. 21-29.
8. A.N. Eden and T. Mudge, "The YAGS Branch Prediction Scheme," *Proc. Int'l Symp. Computer Architecture* (ISCA 25), ACM Press, 1998, pp. 69-77.
9. S. Jourdan et al., "The Effects of Mispredicted-Path Execution on Branch Prediction Structures," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 96), IEEE CS Press, 1996, pp. 58-67.
10. T. Austin et al., "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, no. 2, Feb. 2002, pp. 59-67.
11. J.S. Emer and D.W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proc. Int'l Symp. Computer Architecture* (ISCA 11), ACM Press, 1984, pp. 301-310.
12. K.M. Lepak, H.W. Cain, and M.H. Lipasti, "Redeeming IPC as a Performance Metric for Multithreaded Programs," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 03), IEEE CS Press, 2003, pp. 232-243.

13. J.J. Yi et al., "Characterizing and Comparing Prevailing Simulation Techniques," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE Press, 2005, pp. 266-277.
14. Y. Luo and L.K. John, "Efficiently Evaluating Speedup Using Sampled Processor Simulation," *IEEE Computer Architecture Letters,* 2004, http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/letters/ca/&toc=comp/letters/ca/2004/01/y1toc.xml&DOI=10.1109/L-CA.2004.6.

Alaa R. Alameldeen is a research scientist in the Oregon Microarchitecture Lab at Intel Corporation in Hillsboro, Oregon. His research interests include memory and cache system design for multiprocessors and chip multiprocessors, and performance analysis of multithreaded workloads. He received a BSc and an MSc in computer science from Alexandria University, Egypt, and an MSc and a PhD from the University of Wisconsin-Madison, also in computer science.

David A. Wood is a professor in both the Computer Sciences and Electrical and Computer Engineering departments at the University of Wisconsin-Madison. Along with Mark Hill, he leads the Wisconsin Multifacet project (http://www.cs.wisc.edu/multifacet), which explores techniques for improving the availability, designability, programmability, and performance of commercial multiprocessor servers. His research interests include chip multiprocessors and transactional memory. Wood has a PhD in computer sciences from the University of California, Berkeley. He is a fellow of the IEEE and the ACM, and a member of the IEEE Computer Society.

Direct questions and comments about this article to Alaa R. Alameldeen, Oregon Microarchitecture Lab, Intel Corporation, 2111 NE 25th Ave., M/S JF2-04, Hillsboro, OR 97124; alaa.r.alameldeen@intel.com.