Interaction Cost: For When Event Counts Just Don't Add Up

Interaction cost helps improve processor performance and decrease power consumption by identifying when designers can choose among a set of optimizations and when it's necessary to perform them all.

Brian A. Fields
Rastislav Bodík
University of California at Berkeley

Mark D. Hill
University of Wisconsin at Madison

Chris J. Newburn
Intel

Most performance analysis tasks boil down to finding bottlenecks. In the context of this article, a bottleneck is any event (for example, branch mispredict, window stall, or arithmetic-logic unit (ALU) operation) that limits performance. Bottleneck analysis is critical to an architect's work, whether the goal is tuning processors for energy efficiency, improving the effectiveness of optimizations, or designing a more balanced processor.

Yet, despite its importance, bottleneck analysis methodology has lagged behind processor technology. Although microarchitects have successfully converted the growing supplies of transistors into increased performance, the resulting complexity has made bottleneck analysis much more challenging. Instructions are re-ordered and executed in parallel; processor events such as store-buffer stalls and branch target buffer misses occur simultaneously; and speculative computation is occasionally squashed with control redirected. This complexity and fine-grained parallelism make it difficult to identify what the bottlenecks actually are.

For example, when two cache misses occur in the same cycle, we might need to optimize both to increase performance. What if a multiply and window stall occur simultaneously? Is one of them the *true* bottleneck, or do we again need to optimize both? In general, dozens of events might occur simultaneously in a single cycle on a modern machine. Do we therefore need to optimize all of them to remove the cycle? Or can we get by with just a subset?

The key to answering these questions is understanding how bottlenecks interact in a parallel system. A better understanding of interactions could help us improve the performance of not only microarchitectures but also coarse-grained parallel systems, such as chip multiprocessors. Furthermore, studying interactions helps attack the power wall by making the machine more balanced. In other words, interactions help us find the least power-hungry way to achieve a target performance. This article presents the insights for our analysis methodology, which we discuss more extensively elsewhere.[1,2] The "Related Work" sidebar notes other analysis methodologies for out-of-order processors.

A new bottleneck analysis methodology

To illustrate the power of interaction-based bottleneck analysis, we show how it can help

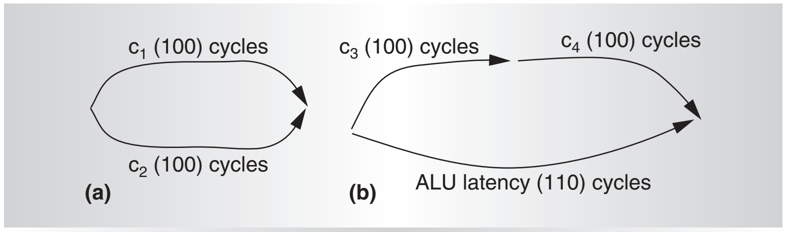

Figure 1. Dependence graphs illustrating the two distinct interaction types that can exist between events. The two cache misses c_1 and c_2 have a parallel interaction, and both would need to be eliminated to improve performance (a). Cache misses c_3 and c_4 experience a serial interaction in which eliminating either cache miss will reduce execution time by 90 cycles, but eliminating both will not improve performance any further (b).

Related work

Several other works have attempted to modify existing analysis methodology for out-of-order processors. ProfileMe is a hardware infrastructure proposal that provides pairwise sampling, which enables computation of some quantitative measures of parallelism.[1] Other work aims to interpret parallelism through fetch[2] and commit attribution,[3,4] and at least one combines attribution with some dependence information.[5] Tune and colleagues first used a dependence graph to compute the cost of individual instructions in a simulator.[6] None of these methodologies has been used to explicitly measure interactions, however, which is our focus.

References

1. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Int'l Symp. Microarchitecture*, IEEE CS Press, 1997, pp. 292-302.
2. S. Patel, M. Evers, and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," *Proc. 25th Int'l Symp. Computer Architecture*, IEEE CS Press, 1998, pp. 262-271.
3. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2002.
4. V.S. Pai, P. Ranganathan, and S.V. Adve, "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology," *Proc. 3rd Int'l Symp. High Performance Computer Architecture*, IEEE CS Press, 1997, pp. 72-83.
5. R. Sasanka, C.J. Hughes, and S.V. Adve, "Joint Local and Global Hardware Adaptations for Energy," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 144-155.
6. E. Tune, D. Tullsen, and B. Calder, "Quantifying Instruction Criticality," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2002, pp. 104-113.

microarchitects designing a machine with a long pipeline. In our case study, we face the problem of long wire delays forcing the level-one cache access latency to be four cycles instead of the expected one or two. This increased latency causes many level-one cache accesses to appear on the microarchitectural critical path. Our goal is to exploit bottleneck interactions to remove some level-one cache accesses from the critical path, even though we cannot reduce the latency.

Before we tackle this problem, let's consider the simple example of two cache misses. As we mentioned earlier, if two cache misses execute in parallel, we must optimize both to improve performance. We call this type of interaction between bottlenecks a *parallel interaction*. But is it the only type of interaction that is possible?

Consider the situation in which two cache misses, each with 100-cycle latency, are data dependent, and parallel to both is a 110-cycle chain of ALU operations. In this example, optimizing both cache misses will not help any more than optimizing only one, because in either case execution time is reduced by the same amount (90 cycles). This effect represents a different type of interaction than the parallel cache misses: in one case you must optimize both, whereas in the other, optimizing only one of the two makes most sense. We call this second interaction type a *serial interaction* because the two misses are in series.

Figure 1 uses dependence graphs to illustrate the two interaction types.

Having discovered two interaction types, it is natural to ask whether more types are possible. Furthermore, how can we make analysis of interactions systematic and quantifiable?

To answer these questions, we first define the performance cost of an event e, denoted $cost(e)$, as the execution time reduction obtained when e is idealized. (In other words, $cost(e)$ is the number of cycles that can be blamed on e.) The meaning of "idealized" depends on the type of event—for example, a cache miss is idealized to a hit, a branch mispredict is made correct, and an ALU operation's latency is set to zero. We can similarly define the cost of a set of events as the execution time reduction when more than one event is idealized.

We are now ready to discover interactions between two events e_1 and e_2 by comparing the sum of the benefit of optimizing the events separately ($cost(e_1) + cost(e_2)$) to the benefit of optimizing them together ($cost(\{e_1, e_2\})$). Only three distinct possibilities exist, one each for

Table 1. Interaction cost measurements on a machine with four-cycle data-cache latency.

Category	gcc	gzip	parser	perl	twolf	vortex
Dl1 (level-one data-cache latency)	18.6	31.9	19.1	31.6	19.1	28.5
Dl1 + win (instruction window stalls)	-3.9	-10.5	-6.6	-5.6	-4.2	-25.9
Dl1 + bw (processor bandwidth: fetch, issue, commit)	11.4	5.6	4.6	9.2	1.5	16.8
Dl1 + bmisp (branch mispredictions)	-6.3	-3.4	-2.4	-7.3	-5.7	-0.2
Dl1 + dmiss (data-cache misses)	-1.4	-1.0	-1.8	-0.2	-2.3	-1.8
Dl1 + shalu (one-cycle ALU operations)	-1.8	-12.7	-4.8	-1.8	-0.4	-4.7
Dl1 + lgalu (multicycle ALU operations)	-0.3	-0.5	-0.1	-0.7	-0.0	-1.3
Dl1 + imiss (instruction cache misses)	0.3	0.0	0.0	1.1	0.0	0.6

Note: We calculated interaction costs as a percent of execution time, using the dependence graph in a simulator. We took the measurements on a six-way out-of-order processor with a 64-entry instruction window.[2] Interaction costs vary significantly across benchmarks, suggesting that an interaction cost characterization of the target workload would be useful early in the chip design process. The variation also indicates that dynamic optimization would be beneficial.

no interaction (that is, independence), parallel interaction, and serial interaction:

- *independent, $cost(\{e_1, e_2\}) = cost(e_1) + cost(e_2)$;*
- *parallel, $cost(\{e_1, e_2\}) > cost(e_1) + cost(e_2)$; and*
- *serial, $cost(\{e_1, e_2\}) < cost(e_1) + cost(e_2)$.*

Thus, we can characterize the performance effects of bottlenecks by considering only two types of interactions (parallel or serial). To inform the optimizer (automatic or human) of the degree of interaction, we define a natural quantitative measure called *interaction cost.* The interaction cost of e_1 and e_2, denoted $icost(\{e_1, e_2\})$, is the difference between the aggregate cost of the two events and the sum of their individual costs:

$$icost\left(\left\{e_1, e_2\right\}\right) \overset{\text{def}}{=} icost\left(\left\{e_1, e_2\right\}\right) - cost\left(e_1\right) - cost\left(e_2\right)$$

Notice the power of *icost*: it characterizes the interaction between events in a single number, with straightforward interpretation. The sign indicates the type of interaction (positive for parallel, negative for serial), and the magnitude indicates the degree of interaction. An interaction cost of zero means the two events are independent and can be optimized separately.

Case study: balancing a long pipeline

Returning to the long pipeline case study, recall that, due to wire delays, the level-one cache access latency is four cycles instead of the expected one or two. As microarchitects, we would like to reduce the increased latency's effect on performance. First, to understand the problem better, we measure and interpret the interaction costs between level-one cache accesses and other machine events.

We can easily compute the interaction cost between two sets of events, S_1 and S_2, by running multiple simulations with the appropriate resources idealized—first obtaining $cost(S_1)$, $cost(S_2)$, and $cost(\{S_1, S_2\})$, and then using the *icost* definition. In our study, we used a more efficient method involving repeated measurement of the critical path length on a graph that models the processor's performance.[1-3] Regardless of how we compute interaction costs, the analysis methodology remains the same.

Table 1 shows the interaction costs involving level-one cache accesses. Data cache accesses have a large singleton (dl1) cost, typically contributing 18 to 32 percent of the execution time. This means that reducing the dl1 latency to zero would eliminate 18 to 32 percent of the execution time.

From Table 1, we see a significant positive interaction cost (parallel interaction) between dl1 and bandwidth (bw)—for example, 11.4 percent for gcc. We interpret this interaction in the same way as the parallel cache misses in Figure 1b: we need to improve dl1 and bw simultaneously to remove 11.4 percent of the cycles (for gcc). Although parallel interactions generally provide the microarchitect with useful knowledge, in this case we cannot reduce the

Table 2. Interaction costs resulting from an increase in instruction window size for Vortex.

	No. of entries		
Category	64	128	256
dl1	28.5	9.8	4.3
Win	39.4	21.3	13.6
dl1 + win	-25.9	-8.14	-2.7
Execution time	100.0	80.8	75.0

latency of dl1, so we cannot remove these cycles.

We also see some negative interaction costs (serial interactions). The largest of these is between dl1 and the instruction window (win)—for example, −25.9 percent for vortex. Are these serial interactions exploitable? Well, recall the effect of the serial interaction between the two cache misses in Figure 1b: we needed to optimize only one of the two misses. The same principle applies here: only one of dl1 or win must be optimized to remove (for vortex) 25.9 percent of the cycles. In other words, the serial interaction lets us choose which resource to optimize. Because in this case we cannot optimize level-one cache latency, we might consider increasing the instruction window size instead.

So, let's explore what happens when we enlarge the instruction window. Table 2 shows the interaction costs obtained when we increase the window's size from 64 to 256 entries (for vortex).

As expected, performance improves substantially, reducing execution time by 25 percent. In addition, although level-one cache latency does not improve at all, its cost decreases considerably with the increased window size. This empirically validates the serial interaction (more detailed validations appear elsewhere[2]).

Although some readers might already be familiar with the relationship between level-one cache accesses and the instruction window, notice how quickly interaction cost analysis discovered the effect. It did not require an expert's analysis; nor did we have to study reams of simulator output. Instead, by following a simple formula we knew which simulations to run, and the interpretation was straightforward and systematic. In fact, interaction cost analysis provides insights in an almost automatic fashion. This expediency could make such analysis valuable as microarchitects explore the undiscovered space of future processor designs.

Other applications of interaction cost

As mentioned earlier, interaction cost has applications beyond microarchitectural performance improvement.

You could use them in a multiprocessor to adjust the speed of the individual processor cores, so no one runs faster than is needed to maintain overall performance—in other words, to keep the system balanced. Interaction cost can also be used to answer energy-related questions. For example, what should you do when a resource A (for example, the instruction window) dominates the power budget, but you can't slow down A without crippling the overall design? The answer is to find another resource B (the data cache, for example) that serially interacts with A. Then, A could be crippled, losing performance but saving power, and B could be sped up (hopefully only slightly) to compensate for the performance loss.

Shotgun profiling

Interaction costs can also provide a way to interpret hardware performance counters on real hardware. With a relatively modest extension to existing hardware counters, a profiler can collect sufficient information to build (offline) a dependence graph of the execution. The dependence graph can then be analyzed to compute costs and from them also interaction costs, providing the same insights on a real machine that can be obtained in a simulator. This technique, called *shotgun profiling*, is named for its similarity to shotgun genome sequencing.[4]

In addition to facilitating the interpretation of collecting data, interaction costs promote simpler communication because they provide a concise language for communicating conclusions. For example, we can state the conclusion of the case study as simply that the level-one accesses and instruction window "interact serially," meaning that improving one makes improving the other less necessary. Previously, we would need a sensitivity study to convey the same idea.

Moreover, interaction cost is a concept applicable not only to the microarchitecture domain, but to any system that exhibits the parallelism interaction cost was designed to analyze—such as distributed applications, cluster computing, and sensor networks. By developing the underlying interaction cost methodology, we can open up new opportunities for performance understanding and optimization in many areas of computer design. MICRO

Acknowledgements

We thank David Patterson for comments on this article, as well as those acknowledged in the original work.

References

1. B.A. Fields et al., "Using Interaction Costs for Microarchitectural Bottleneck Analysis," *Proc. 36th Int'l Symp. Microarchitecture*, IEEE CS Press, 2003, pp. 228-239.
2. B.A. Fields et al., "Interaction Cost and Shotgun Profiling," *ACM Trans. Architecture and Compiler Optimization*, vol. 1, no. 3, 2004, pp. 272-304.
3. B. Fields, S. Rubin, and R. Bodík, "Focusing Processor Policies via Critical-Path Prediction," *Proc. 28th Int'l Symp. Computer Architecture*, ACM Press, 2001, pp. 74-85.
4. R.D. Fleischmann et al., "Whole-Genome Random Sequencing and Assembly of Haemophilus-Influenzae," *Science*, vol. 269, 1995, pp. 496-512.

Brian Fields is a PhD student at the University of California, Berkeley, where he is the student leader of the BAFL project (http://www.cs.berkeley.edu/~bfields/bafl). His research interests include performance analysis and developing next-generation microarchitectures. He has a MS in computer science from the University of Wisconsin, Madison, and a BS in computer engineering from the University of Cincinnati.

Rastislav Bodík is an assistant professor of computer science at the University of California, Berkeley, where he co-leads the BAFL project with Mark Hill. His research interests range from program analysis and software engineering tools to computer architecture. He has a PhD from the University of Pittsburgh.

Mark D. Hill is a professor in both the computer sciences department and the electrical and computer engineering department at the University of Wisconsin, Madison, where he coleads the BAFL project with Ras Bodik and the Wisconsin Multifacet project (http://www.cs.wisc.edu/multifacet/) with David Wood. His research interests include cache design, cache simulation, translation buffers, memory consistency models, parallel simulation, and parallel computer design. He has a PhD from University of California, Berkeley. He is an ACM Fellow and a Fellow of the IEEE.

Chris (CJ) Newburn is part of the IA32 architecture team in the Desktop Product Group at Intel. His research interests lie in feedback-directed optimization, managed runtimes, performance analysis, parallelization, virtualization, and enhancements to memory systems. He has a PhD in electrical and computer engineering from Carnegie Mellon University. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Brian Fields, 517 Soda Hall, Computer Science Division, #1776, Univ. of California, Berkeley, Berkeley, CA 94720-1776; bfields@cs.berkeley.edu.