

# Interaction Cost: For when event counts just don't add up

Final version for IEEE Micro Top Picks '04

Brian A. Fields<sup>1</sup> Rastislav Bodík<sup>1</sup> Mark D. Hill<sup>2</sup> Chris J. Newburn<sup>3</sup>

<sup>1</sup>University of California-Berkeley <sup>2</sup>University of Wisconsin-Madison <sup>3</sup>Intel Corporation

## Abstract

FINE-GRAIN CONCURRENCY IN MODERN PROCESSORS MAKES PERFORMANCE UNDERSTANDING CHALLENGING. INTERACTION COST HELPS IMPROVE PERFORMANCE AND DECREASE POWER CONSUMPTION BY IDENTIFYING WHEN DESIGNERS MAY CHOOSE AMONG A SET OF OPTIMIZATIONS OR MUST DO ALL OF THEM.

Most performance analysis boils down to finding bottlenecks. In our context, a bottleneck is any event (*e.g.*, branch mispredict, window stall, or ALU operation) that limits performance. Bottleneck analysis is critical for 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. While increasing supplies of transistors have been successfully converted into increasing 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 BTB misses occur at the same time; and speculative computation is occasionally squashed with control redirected. This complexity and fine-grain parallelism makes it difficult to know what the bottlenecks actually are.

For example, when two cache misses occur in the same cycle, we may need to optimize both to increase performance. How about 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, there may be dozens of events occurring simultaneously in a single cycle on a modern machine. Does that mean we 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* with one another in a parallel system. The study of interactions is applicable to improving the performance of not only microarchitectures but also coarse-grained parallel systems (*e.g.*, chip multiprocessors). Furthermore, studying interactions helps attack the power wall by making the machine more *balanced*. Or, in other words, interactions help find the least power-hungry way to achieve a target performance. Below we provide the insights for the analysis methodology more extensively presented elsewhere [3, 4].

## 1 Interaction Costs: A New Bottleneck Analysis Methodology

To illustrate the power of interaction-based bottleneck analysis, we show how it can help microarchitects when designing a machine with a long pipeline. In the 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 the latency cannot be reduced.

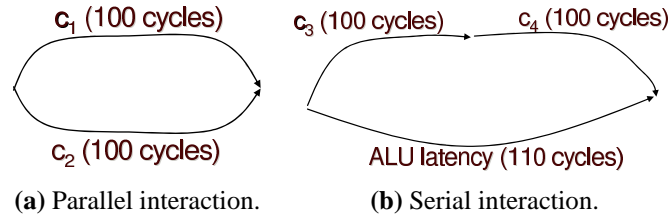


Figure 1: **Two types of interactions.** Dependence graphs can conveniently illustrate the two distinct types of interactions that can exist between events: *parallel* and *serial*. In (a), the two cache misses  $c_1$  and  $c_2$  have a parallel interaction. Both cache misses would need to be eliminated to improve performance. In (b),  $c_3$  and  $c_4$  experience a serial interaction. Here, eliminating either cache miss will reduce execution time by 90 cycles, but eliminating both will not improve performance any further.

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

Well, consider the situation where two cache misses, each with 100 cycle latency, are data-dependent, and in 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, since 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, in the other it makes most sense to optimize only one of the two. Since in this example the two misses are in series, we call the effect a *serial interaction*. It is often convenient to visualize interactions using dependence graphs, like the ones in Figure 1.

Now that we have discovered two types of interactions, 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 both separately ( $cost(e_1) + cost(e_2)$ ) to the benefit of optimizing both together ( $cost(\{e_1, e_2\})$ ). There are only three distinct possibilities, one each for no interaction (*i.e.*, independence), parallel interaction, and serial interaction:

$$\begin{aligned}
 cost(\{e_1, e_2\}) &= cost(e_1) + cost(e_2) \Leftrightarrow \textit{Independent} \\
 cost(\{e_1, e_2\}) &> cost(e_1) + cost(e_2) \Leftrightarrow \textit{Parallel Interaction} \\
 cost(\{e_1, e_2\}) &< cost(e_1) + cost(e_2) \Leftrightarrow \textit{Serial Interaction}
 \end{aligned}$$

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 defined as the difference between the aggregate cost of the two events and the sum of their individual costs:

Category	gcc	gzip	parser	perl	twolf	vortex
<b>dl1</b>	<b>18.6</b>	<b>31.9</b>	<b>19.1</b>	<b>31.6</b>	<b>19.1</b>	<b>28.5</b>
<b>dl1+win</b>	<b>-3.9</b>	<b>-10.5</b>	<b>-6.6</b>	<b>-5.6</b>	<b>-4.2</b>	<b>-25.9</b>
dl1+bw	11.4	5.6	4.6	9.2	1.5	16.8
<b>dl1+bmiss</b>	<b>-6.3</b>	<b>-3.4</b>	<b>-2.4</b>	<b>-7.3</b>	<b>-5.7</b>	<b>-0.2</b>
dl1+dmiss	-1.4	-1.0	-1.8	-0.2	-2.3	-1.8
<b>dl1+shalu</b>	<b>-1.8</b>	<b>-12.7</b>	<b>-4.8</b>	<b>-1.8</b>	<b>-0.4</b>	<b>-4.7</b>
dl1+lgalu	-0.3	-0.5	-0.1	-0.7	-0.0	-1.3
dl1+imiss	0.3	0.0	0.0	1.1	0.0	0.6

Table 1: **Interaction cost measurements on a machine with four-cycle data-cache latency.** *icosts* are presented here as a percent of execution time and were calculated using the dependence graph in a simulator. The categories are: 'dl1' → level-one data cache latency; 'win' → instruction window stalls; 'bw' → processor bandwidth (fetch,issue,commit bandwidths); 'bmiss' → branch mispredictions; 'dmiss' → data-cache misses; 'shalu' → one-cycle integer operations; 'lgalu' → multi-cycle integer and floating-point operations; and 'imiss' → instruction cache misses. The measurements were taken on a 6-way out-of-order processor with a 64-entry instruction window [4]. Note that *icosts* vary significantly across benchmarks, suggesting an *icost* characterization of the target workload would be useful early on in the chip design process. Also, the variation makes it seem likely that dynamic optimization would be beneficial.

$$icost(\{e_1, e_2\}) \stackrel{\text{def}}{=} cost(\{e_1, e_2\}) - cost(e_1) - cost(e_2)$$

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 *icost* of zero means the two events are independent and can be optimized separately.

## 1.1 Case Study: Balancing a Long Pipeline

Now that we have a metric for quantifying interactions, let's explore the promised 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 performance effect of this increased latency. The first step is to understand the problem better, by measuring and interpreting the *icosts* between level-one cache accesses and other machine events.

Note that the *icost* between two sets of events,  $S_1$  and  $S_2$ , can be computed easily 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 above. We employed a more efficient method involving repeated measurement of the critical path length on a graph that models the performance of the processor [2–4]. Regardless of how *icosts* are computed, the analysis methodology remains the same.

The interaction costs involving level-one cache accesses are shown in Table 1. Notice first that data-cache accesses have a large singleton (*dl1*) cost, typically contributing 18–32% of the execution time. This means that 18–32% of the execution time would be eliminated if the *dl1* latency was reduced to zero.

From Table 1, we see a significant positive *icost* (parallel interaction) between *dl1* and bandwidth (*bw*), e.g., 11.4% for *gcc*. This interaction should be interpreted in the same way as the parallel cache misses of Figure 1(b): both *dl1* and *bw* need to be improved simultaneously in order to remove 11.4% of the cycles (for *gcc*). While, in general, parallel interactions provide useful knowledge to the microarchitect, in this case we cannot reduce the latency of *dl1*, so we cannot remove these cycles.

We also see some negative *icosts* (serial interactions), the largest in magnitude being between *dl1* and the instruction window (*win*), e.g., -25.9% for *vortex*. Could these serial interactions be exploited? Well, recall the effect of the serial interaction between the two cache misses in Figure 1(b): only one of the two misses needed to be optimized. The same principle applies here: only one of *dl1* and *win* needs to be

optimized to remove (for *vortex*) 25.9% of the cycles. In other words, the serial interaction gives us a *choice* of which resource to optimize. Since, in this case, we cannot optimize level-one cache latency, we might consider increasing the size of the instruction window instead.

So, let's explore what happens when the instruction window is enlarged. The *icosts* obtained when its size is increased from 64 to 256 is shown below (for *vortex*).

	vortex		
	64	128	256
dll	28.5	9.8	4.3
win	39.4	21.3	13.6
<b>dll+win</b>	<b>-25.9</b>	<b>-8.14</b>	<b>-2.7</b>
<i>Exe Time</i>	<i>100.0</i>	<i>80.8</i>	<i>75.0</i>

First note that, as expected, performance improves substantially, reducing execution time by 25%. Also notice that despite not improving level-one cache latency at all, its cost decreases considerably with the increased window size. This is an empirical validation of the serial interaction (more detailed validations appear in Fields, et al. [4]).

Although this particular relationship between level-one cache accesses and the instruction window may already be known to some, notice how quickly *icost* analysis discovered the effect. No hard thinking by an expert or studying reams of simulator output was required. Instead, a simple formula was followed to know what simulations to run, and the interpretation was straightforward and systematic. In fact, *icost* analysis provides insights in an almost automatic fashion. This expediency could make such analysis very valuable as microarchitects explore the undiscovered space of future processor designs.

## 1.2 Other Applications of Icosts

As mentioned above, interaction cost has applications beyond microarchitectural performance improvement. In a multiprocessor you could use them to adjust the speed of the individual processor cores, so that 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, such as: What should one do when the power budget is dominated by a resource *A* (e.g., the instruction window), but you can't slow down *A* without crippling the overall design? The answer is to find another resource *B* (e.g., the data cache) that serially interacts with *A*. Then, *A* could be crippled, losing performance but saving power, and *B* could be sped up (hopefully only slightly) compensating for the performance loss.

## 1.3 Shotgun Profiling

We have shown that *icosts* can be useful in a simulator for microarchitects designing machines, but they also provide a way to interpret hardware performance counters on real hardware. With a relatively modest extension to existing hardware counters, sufficient information can be collected to build (offline) a dependence graph of the execution. Then, the dependence graph can be analyzed to compute costs and from them also interaction costs, providing the same insights on real machine that can be obtained in a simulator. This technique is called *shotgun profiling*, named for its similarity to shotgun genome sequencing [5].

## 2 Related Work

Several other works have attempted to modify existing analysis methodology to be suitable for out-of-order processors. ProfileMe [1] is a hardware infrastructure proposal that provides “pair-wise” sampling, which enables computation of some quantitative measures of parallelism. Other work aims to interpret parallelism through fetch (e.g., [8]) and commit attribution (e.g., [6, 7]), and at least one that combines attribution with some dependence information [9]. Another work, by Tune et al. [10], was the first to use a dependence

graph to compute the cost of *individual* instructions in a simulator. None of these methodologies have been used to explicitly measure interactions, however, which is the focus of this article.

### 3 Conclusion

We have presented an analysis methodology based on *interaction costs* that is suitable for modern, fine-grain parallel machines, such as out-of-order processors. Interaction costs provide *easy interpretation*, since there is a simple formula on how to interpret collected data, based on the sign and magnitude of the *icosts*. Further, interaction costs promote *simpler communication*, since 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, a sensitivity study would be needed to convey the same idea.

**Acknowledgements.** We thank David Patterson for comments on this paper, as well as those acknowledged in the original work [4].

### References

- [1] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *30<sup>th</sup> International Symposium on Microarchitecture*, Dec 1997.
- [2] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *28<sup>th</sup> International Symposium on Computer Architecture*, Jun 2001.
- [3] B. A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *36<sup>th</sup> International Symposium on Microarchitecture*, Dec 2003.
- [4] B. A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Compiler Optimization*, 2004.
- [5] R. D. Fleischmann et al. Whole-genome random sequencing and assembly of haemophilus-influenzae. *Science*, 269:496–512, 1995.
- [6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 3<sup>rd</sup> edition, 2002.
- [7] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *3<sup>rd</sup> International Symposium on High Performance Computer Architecture*, Feb 1997.
- [8] S. Patel, M. Evers, and Y. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *25<sup>th</sup> International Symposium on Computer Architecture*, Jun 1998.
- [9] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In *10<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.
- [10] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In *11<sup>th</sup> International Conference on Parallel Architectures and Compilation Techniques*, Sep 2002.