Using Interaction Costs for Microarchitectural Bottleneck Analysis

Brian A. Fields[1] Rastislav Bodík[1] Mark D. Hill[2] Chris J. Newburn[3]
[1]University of California-Berkeley [2]University of Wisconsin-Madison [3]Intel Corporation

Abstract

Attacking bottlenecks in modern processors is difficult because many microarchitectural events overlap with each other. This parallelism makes it difficult to both (a) assign a cost to an event (*e.g.,* to one of two overlapping cache misses) and (b) assign blame for each cycle (*e.g.,* for a cycle where many, overlapping resources are active). This paper introduces a new model for understanding event costs to facilitate processor design and optimization.

First, we observe that everything in a machine (instructions, hardware structures, events) can interact in only one of two ways (in parallel or serially). We quantify these interactions by defining *interaction cost*, which can be zero (independent, no interaction), positive (parallel), or negative (serial).

Second, we illustrate the value of using interaction costs in processor design and optimization.

Finally, we propose performance-monitoring hardware for measuring interaction costs that is suitable for modern processors.

1 Introduction

Modern microprocessors achieve much of their performance through rigorous exploitation of fine-grain parallelism. The key dilemma caused by this parallelism is, *Which event are we to blame for a cycle that experienced two (or more) simultaneous events (for example, when a window stall and a multiplication occurred simultaneously)?* Clearly, both of these events must be optimized to remove the cycle, but how do we express this fact in a performance breakdown?

Another view of the overlap dilemma is to ask, *What performance monitoring hardware can I add to my processor to answer these questions?* Counting events, event latencies, or both also fails to capture overlap.

This paper argues that if we could answer the above questions without losing track of the microarchitectural parallelism, we would help the designer to resize just the right queue, predict the most critical dependence, or, conversely, economically reduce the sizes of non-bottleneck resources, saving area and energy. In short, we could build more balanced machines, where no resource is waiting on another.

We answer these questions with performance analysis that is simple, yet powerful enough to make sense out of simultaneous bottlenecks in complex machines. A *bottleneck* is any *set* of events that contribute to execution time, while the *cost* of a bottleneck is simply the speedup obtained from idealizing the bottleneck's events. How events are grouped into a *set* depends on the application of the analysis. For example, a software prefetching optimization might consider the set of events consisting of all cache misses from a single static load, while hardware designers might focus on all events pertaining to a resource (*e.g.,* all branch mispredictions).

Cost is a powerful metric because it reveals how much an optimization helps before further improvement is stopped by a secondary bottleneck. Moreover, events with cost zero may be good targets for "de-optimization" (*e.g.,* making a queue smaller without affecting performance).

This standard notion of cost, of course, tells us nothing about our simultaneous bottlenecks, as illustrated by the fact that the cost of each of two completely parallel cache misses is zero. As the first contribution of our paper, we define *interaction cost* (*icost*) which reveals how two (or more) events interact in a (parallel) microexecution. Specifically, interaction cost of two events a and b is the difference in speedup between idealizing both together ($cost(a, b)$) and the sum of idealizing them individually: $icost(a, b) \overset{\text{def}}{=} cost(a, b) - cost(a) - cost(b)$. That is, interaction cost quantifies the cycles that can be removed only by optimizing both events together. Analogously, we can define the interaction cost between sets of events (*e.g.,* all cache misses interacting with all ALU operations) by replacing a and b with sets of events.

The second contribution of our paper is to explore the utility of interaction cost for everyday design practice. We find that, somewhat surprisingly, interaction costs can be zero (*e.g.,* for two independent cache misses), positive (*e.g.,* for two parallel cache misses), and even negative (*e.g.,* for two cache misses in series with each other but in parallel with other events).

A *zero interaction* cost between two (sets of) events implies that we can design and evaluate optimizations for the two in isolation, as the events are independent: optimizing one will not change the cost of the other.

A *parallel interaction* (*i.e.,* positive $icost$) reveals that events overlap, which implies that there is speedup which can be gained only by optimizing both events (*e.g.,* two cache misses that completely overlap).

A *serial interaction* (*i.e.,* negative $icost$) means that two events are in series with each other, but also in parallel with some other event. It thus reveals that completely optimizing both events is not worthwhile; rather, one should target either only one or both partially. Serial interaction gives the designer flexibility to attack what is easiest to improve and eschew optimizing structures that are already too big, power-hungry, or complex.

Costs and interaction costs are most useful in practice if they can be efficiently measured in both simulation and hardware (*e.g.,* with an extension to performance counters). They can obviously be computed by running many idealized and unidealized simulations. This approach, however, requires 2^n simulations for n events or resources, which may be too expensive if n is large.

For greater efficiency, we perform manipulations on a microexecution dependence graph as an alternative to complete resimulation. This graph is similar to the one used in previous work [11,12,37]. It captures both architectural dependencies (*e.g.,* data dependencies) and microarchitectural events (*e.g.,* branch mispredictions).

Finally, to measure interaction costs on real hardware running "live" workloads, we show, as our third contribution, how hardware can sample an execution in sufficient detail to construct a statistically representative microarchitecture graph. We call this hardware a *shotgun profiler*, because of its similarity to shotgun genome sequencing [14]. The profiler has low complexity (of the order of ProfileMe [9]) and is suitable not only for measuring interaction costs, but also for accurately computing the simple individual costs. Thus, it may serve as an alternative to the current hard-to-interpret performance counters.

2 Icost: Unifying notion of performance analysis

As motivated in the introduction, determining the *costs* and *interaction costs* of events is essential to many forms of performance analysis. By defining interaction costs, this section deals with the effects of microarchitectural parallelism on the cost of events. To achieve uniform analysis, we use the term *event* to refer to any stall cause, whether due to data dependences, resource constraints, or microarchitectural events.

2.1 Cost

Intuitively, the cost of an event is not its execution latency, but its contribution to the overall execution time of the program. Equivalently, the cost is the execution time decrease obtained if the event is *idealized*. Table 1 lists how some events can be idealized. Let e be an event, t be base execution time (nothing idealized), and $t(e)$ be execution time with e idealized. We formally define the cost of e, $cost(e)$ as

$$cost(e) \overset{\text{def}}{=} t - t(e)$$

The cost of an event can be naturally generalized to an *aggregate* cost of a set of dynamic events S. This allows us to compute, for example, the cost of a cache as the total speedup when all cache misses are idealized.

Event type	**How to idealize in a simulator**
Icache, Dcache misses	Turn misses into hits
ALU operation	Give ALU zero cycle latency
Fetch,Issue,Commit BW	Use infinite BW
Branch mispredict	Turn mispredicts into correct preds
Instruction window	Use infinite window

Table 1: **Idealizing events.** Listed are techniques to idealize a few of the events studied in this paper. Due to practical constraints (finite memory), we approximate an infinite window by using one that is twenty times larger than the baseline.

Observing the idealizations of Table 1 clarifies why this definition of cost is useful. A compiler seeking to prefetch load instructions would want to know how much execution time would improve if all dynamic cache misses from a single static load were idealized to hits. A hardware value predictor would want to know the improvement from idealizing particular data dependences. Finally, an architect considering enhancements to the instruction window would like to know how much such enhancements could improve performance.

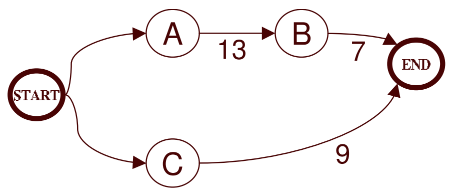
2.2 Interaction Cost

While knowing the costs of individual events is useful, they are not always sufficient to drive optimization decisions. For instance, two completely parallel cache misses (c_1 and c_2) both have cost of zero ($cost(c_1)$ = $cost(c_2)$ = 0), since idealizing one would leave the overall critical path length unchanged. Nevertheless, prefetching both loads may have substantial benefit.

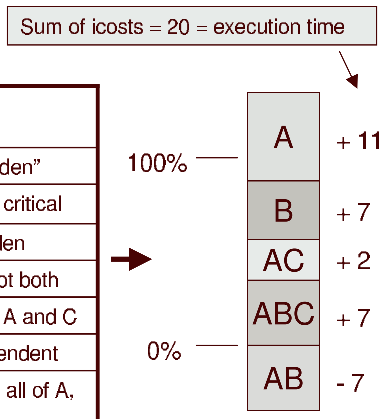
Similar scenarios occur with analyses for making microarchitectural design decisions. For instance, an architect may find, via idealization, that the cost of cache load ports is low, suggesting it is not worthwhile to make the cache dual-ported. The reality may be, however, that if the instruction window is also enlarged, increasing cache bandwidth could provide significant gain.

Essentially, the problem is that measuring the cost of individual events is only useful for determining "how critical" a *single* event is. In other words, standard cost gives no information about the content of "secondary" critical paths. While quantifying all secondary paths may seem a daunting task, we show below how to get a handle on the problem by measuring *interactions* between individual event costs.

Consider, for instance, the above example of the two cache misses. While the cost of the individual cache misses are zero, the *aggregate* cost of both cache misses, obtained by measuring the execution time reduction from idealizing both c_1 and c_2 simultaneously, would be large. By knowing this aggregate cost, denoted $cost(\{c_1, c_2\})$, the program optimizer would know that while prefetching only one load would give little benefit, prefetching both would give significant benefit. We term this phenomenon, where $cost(\{c_1, c_2\}) > cost(c_1) + cost(c_2)$, a *parallel interaction*.



Cat. Name	Expression for computing icost	Interaction Cost (icost)	Comments
A	cost(A)	11	A is partially "hidden"
B	cost(B)	7	All of B's latency is critical
C	cost(C)	0	C is totally hidden
AB	cost({A,B}) - icost(A) - icost(B)	11 -11 - 7 = -7	Optimize A or B, not both
AC	cost({A,C}) - icost(A) - icost(C)	13 - 11 - 0 = 2	Bonus for optimizing A and C
BC	cost({B,C}) - icost(B) - icost(C)	7 - 7 - 0 = 0	B and C are independent
ABC	cost({A,B,C}) - icost({A,B}) - icost({A,C}) - icost({B,C}) - icost(A) - icost(B) - icost(C)	20 - (-7) - 2 - 0 - 11 - 7 - 0 = 7	Bonus for optimizing all of A, B, and C



(a) Breakdown example **(b) One possible visualization**

Figure 1: **Correctly reporting breakdowns.** **(a)** The traditional method for reporting breakdowns does not accurately account for *all* execution cycles, since it attempts to assign blame for each cycle to a *single* event when sometimes multiple events are simultaneously responsible. We propose a new method that uses *interaction costs*, discussed in Section 2.2. In our method, each category corresponds to an interaction cost of a set of "base" categories. **(b)** One possible compact visualization of this breakdown is shown. Here the positive interaction costs cause the stacked-bar chart to extend above 100%, but this is offset by negative interactions – which are plotted below the axis.

Perhaps less intuitively, it is also possible for the opposite parallelism-induced effect to occur, where $cost(\{c_1, c_2\}) < cost(c_1) + cost(c_2)$. One example is if two *dependent* cache misses, each with 100 cycle latency, both occurred in parallel with 100 cycles of ALU operations. In this situation, prefetching both provides no more benefit than prefetching either one alone, implying that a program optimizer would save overhead by performing only one prefetch. We call this phenomenon a *serial interaction*, since the two interacting cache misses occur in series.

In summary, for two events e_1 and e_2:

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

As our paper empirically shows, interactions are common phenomena (after all, there is potential for interaction any time two events occur simultaneously). To inform the optimizer (automatic or human) of the "degree" of interaction, we define interaction cost. Let e_1 and e_2 be two events and $cost(\{e_1, e_2\})$ be the aggregate cost of both events. Then, 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:

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

Thus, for a parallel interaction, $icost(\{e_1, e_2\})$ is the number of extra cycles an optimization that targets both events, instead of just one, could ever hope to benefit. In contrast, for a serial interaction, $icost(\{e_1, e_2\})$ would be negative, reducing the expectation for performance improvement from targeting both events.

The interaction cost of two sets of events, S_1 and S_2, is defined similarly, by replacing e_1 and e_2 with S_1 and S_2 in the above equation. Moreover, the interaction cost of more than two events (or sets) can be defined recursively. Formally, let $\mathcal{P}(U) \setminus U$ denote the proper power set of a set of events U (*i.e.,* all subsets of U except for U itself). Then the interaction cost of U is defined as the cost of U minus the interaction cost of each proper subset of U:

$$icost(\{\}) \overset{\text{def}}{=} 0$$
$$icost(U) \overset{\text{def}}{=} cost(U) - \sum_{V \in \mathcal{P}(U) \setminus U} icost(V)$$

Finally, if U is the set of *all* events in an execution it follows that total execution time always equals the sum of the $icosts$ for the powerset of U. This implies that completely accounting for execution time requires all interaction costs to be considered.

Interaction cost is a valuable tool for analyzing parallelism in out-of-order processors (and, potentially, parallel systems in general). Guiding load-prefetching decisions is only one example. The next section describes how to use interaction costs to construct parallelism-aware performance breakdowns, useful in making architectural design decisions.

New addition	Description
New nodes	Expands the nodes per instruction from three (D for "dispatch into window", E for "execute", and C for "committing") to five (adding R representing "ready to execute" and P representing "completed execution") to provide us with the granularity to model new constraints, which are modeled as edges.
Modeling of fetch/commit BW	Explicit modeling of fetch and commit bandwidth with dependence edges (as opposed to implicitly, with latency placed on DD and CC edges). Specifically, the new model places edges from D_i to D_{i+fbw} and C_i to C_{i+cbw}, where fbw (cbw) are the maximum number of instructions that can be fetched (committed) in a given cycle. This new model does a better job at modeling the effect of an idealization since the new edges are guaranteed to have the same latency before and after the idealization.
Cache-block sharing	Modeling of *cache-block sharing* between loads by placing an edge from the P node of any cache-missing load a to the P node of any subsequent load instruction b that accesses the same cache line. This dependence prevents instruction b from *completing* execution until the cache miss is serviced by a. In this way, we accurately model the effect of partial cache misses: if a is sped up due to an idealization, b may effectively change from a partial miss into a hit.

Table 2: **New additions to graph model over previous work [11, 12, 37].**

name	constraint modeled	edge	
DD	In-order dispatch	$D_{i-1} \rightarrow D_i$	
FBW	Finite fetch bandwidth	$D_{i-fbw} \rightarrow D_i$	where fbw is the maximum no. of insts. fetched in a cycle
CD	Finite re-order buffer	$C_{i-w} \rightarrow D_i$	$w =$ size of the re-order buffer
PD	Control dependence	$P_{i-1} \rightarrow D_i$	inserted if $i-1$ is a mispredicted branch
DR	Execution follows dispatch	$D_i \rightarrow R_i$	
PR	Data dependences	$P_j \rightarrow R_i$	inserted if instruction j produces an operand of i
RE	Execute after ready	$R_i \rightarrow E_i$	
EP	Complete after execute	$E_i \rightarrow P_i$	
PP	Cache-line sharing	$P_j \rightarrow P_i$	inserted if inst. j produces cache miss to block loaded by i
PC	Commit follows completion	$P_i \rightarrow C_i$	
CC	In-order commit	$C_{i-1} \rightarrow C_i$	
CBW	Commit BW	$C_{i-cbw} \rightarrow C_i$	where cbw is the maximum no. of insts. committed in a cycle

Table 3: **Constraints captured by the out-of-order processor performance model.** The meaning of the nodes are as follows: D, instruction dispatch into window; R, all data operands ready but waiting on functional unit; E, executing; P, completed execution; C, committing. The constraints correspond to dependence edges in the graph. Operations are represented by latencies on the edges. An example instance of the dependence graph is shown in Figure 2.

2.3 Applying icost: Parallelism-aware Breakdowns

A **performance breakdown** of a microexecution answers the question, "how much do particular processor resources contribute to overall execution time?" Stated another way, a breakdown is a function that maps each cycle of execution to the events that are responsible for it. By allocating cycles among *base categories* of events (*e.g.,* cache misses, ALU latencies, and the rest), a breakdown accounts for all cycles in the execution.

Traditional performance breakdowns (*a.k.a.,* CPI breakdowns) map each cycle of execution delay to exactly one cause. This is fundamentally not possible in an out-of-order processor, because sometimes multiple causes are to blame for a cycle. As a result, a traditional breakdown cannot accurately account for all cycles.

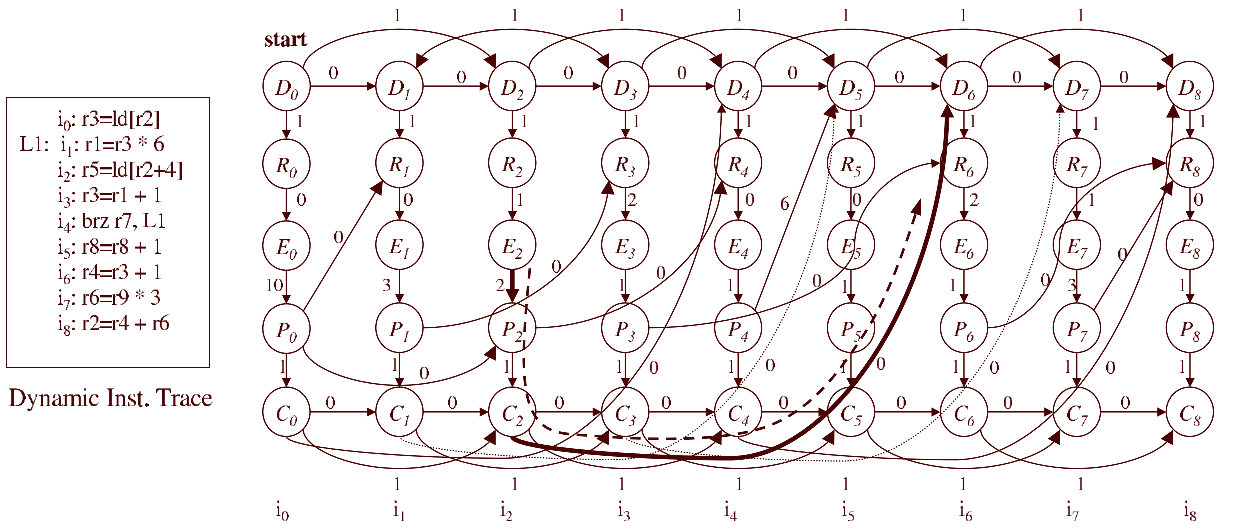
We improve traditional breakdowns by providing information about secondary critical paths. This approach enables an architect to determine when improving multiple resources will yield more benefit than an individual resource. Our solution is to have an explicit *interaction category* for each possible overlap among *base categories*. For example, if the base categories are data-cache misses (*dmiss*), alu operations (*alu*), and branch mispredicts (*bmisp*), then there would be four interaction categories: *dmiss+alu*, *dmiss+bmisp*, *alu+bmisp*, *dmiss+alu+bmisp*. Each category would correspond to an interaction cost, similar to the example of Figure 1. With this representation, it is possible for a breakdown to account for all execution time. Also, while a table is sufficient to completely report a breakdown, graphical visualizations could also be used, such as the stacked-bar chart in Figure 1b.

3 Measuring cost on a dependence graph

Computing all costs and interaction costs for n sets (classes) of events can be done via 2^n simulations. Even if only interaction pairs are measured, a quadratic number of simulations is required. Thus, a more efficient methodology than simulation is desired. Besides this, running multiple idealized simulations may not be possible for performance analysis on real hardware.

Our solution is to determine the *effect* of an idealization without actually performing the idealization. We do this with a dependence-graph model of the microexecution where all the important events and resource constraints are modeled as latency-labeled edges. Then, for each idealization, we only need to alter a bottleneck's edges: by changing their latencies or by removing them.

The dependence graph model. For our purposes, the graph model should meet two requirements: (1) idealiz-



```
    i0: r3=ld[r2]
L1: i1: r1=r3 * 6
    i2: r5=ld[r2+4]
    i3: r3=r1 + 1
    i4: brz r7, L1
    i5: r8=r8 + 1
    i6: r4=r3 + 1
    i7: r6=r9 * 3
    i8: r2=r4 + r6
```

Dynamic Inst. Trace
start
D0 D1 D2 D3 D4 D5 D6 D7 D8
R0 R1 R2 R3 R4 R5 R6 R7 R8
E0 E1 E2 E3 E4 E5 E6 E7 E8
P0 P1 P2 P3 P4 P5 P6 P7 P8
C0 C1 C2 C3 C4 C5 C6 C7 C8
i0 i1 i2 i3 i4 i5 i6 i7 i8

Figure 2: **An instance of the dependence-graph model from Table 3.** The dependence graph represents a sequence of dynamic instructions, assuming a machine with a four-instruction ROB and two-wide fetch/commit bandwidth. The dashed arrow shows how some load access *EP* edges and *CD* window edges are in series and, thus, have the potential to interact serially (see Section 4.1). Note that some other *EP* and *CD* edges are in parallel, thus there is also potential for parallel interaction between loads and the finite window constraint.

ing on the graph should give the same speedup as in the simulator and (2) the analysis should be reasonably efficient. We used a model that provides a level of detail that reasonably meets both requirements (see Section 6 for an empirical assessment of its accuracy and the end of Section 4 for a discussion of efficiency). The model modestly refines previous work [11,12,37] in three ways, as discussed in Table 2. Table 3 describes the nodes and edges; and Figure 2 shows an instance of the model on a sample code snippet.

Measuring cost using the graph. We compute interaction cost with the same post-mortem algorithm that was used to compute individual event cost in Tune, et al. [37]. Their algorithm works by comparing the critical-path lengths of the baseline and idealized graphs – with some optimizations for efficiency. It can be used because, as you recall from Section 2.2, the interaction cost of two events *icost*(a,b) is computed from several simple cost measurements: *cost*(a,b), *cost*(a), and *cost*(b). In general, the *icost* of n events can be computed with $2^n - 1$ *cost* measurements.

4 Icost Tutorial: Optimizing a long pipeline

Several recent studies have found significant performance improvements possible by increasing the length of the processor pipeline. The improvement comes from increased clock frequency, but this improvement is unfortunately offset by the increasing latency of *performance-critical loops*. A loop is a feedback path in the pipeline, where the result of one stage is needed by an earlier stage. Three of the most critical loops include: (i) the latency of a level-one data cache access, (ii) the latency to issue back-to-back operations (the issue-wakeup loop), and (iii) branch mispredictions [2, 15, 17, 31].

In this section, we present a tutorial on using interaction costs, by showing how they can quickly provide insights into processors with long pipelines. Interaction costs show us how to mitigate the performance impact of critical loops. Finally, we compare our icost analysis conclusions to those of a conventional sensitivity study.

4.1 The level-one data cache access loop

Let's assume that the circuit designers optimized the level-one data cache access as much as possible, but nonetheless the latency was higher than expected, say four cycles instead of the typical one or two. The question now is: What is the *most effective* way to change the microarchitecture to mitigate the effect of the high latency? Would it help to: (a) enlarge the branch predictor; (b) increase the number of load ports; (c) increase the data cache size; or (d) increase the fetch bandwidth? Certainly these changes will reduce the cost of each of these resources (if they were on the critical path), but will they also reduce the cost of data cache accesses?

In our case study, before computing the interaction costs, we hypothesized what the outcome of the analysis could be, which amounted to predictions of where *serial* interactions would occur. If a class of microarchitectural events serially interact with data-cache accesses, attacking that resource will also help "hide" the data-cache latency, thereby reducing its performance cost.

We thought data dependences between data-cache missing loads or ALU operations with data-cache accesses (level-one hits) might cause such a serial interaction. Another possibility would be an interaction between branch mispredicts and data-cache accesses, since loads often feed branches. It was difficult, however, to

Category	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
dl1	**22.2**	**24.2**	**18.2**	**13.5**	**18.3**	**30.5**	**7.7**	**19.0**	**31.6**	**19.4**	**28.8**	**19.7**
win	16.4	15.1	15.7	41.0	13.6	23.0	4.2	17.3	4.4	25.1	47.1	23.2
bw	4.4	8.0	7.7	2.8	8.2	5.7	0.5	2.9	8.6	3.9	5.3	5.8
bmisp	41.0	28.6	15.8	12.3	26.3	25.8	26.9	16.5	38.0	24.1	1.9	24.9
dmiss	23.8	7.1	0.7	23.5	26.3	7.7	81.0	32.9	1.4	34.4	21.8	33.7
shortalu	9.9	11.4	5.4	13.8	5.1	20.4	1.4	19.7	7.3	7.8	4.9	7.6
longalu	0.3	0.9	11.8	5.6	0.4	0.7	0.0	0.1	0.8	4.2	1.6	3.6
imiss	0.0	0.7	7.8	0.7	2.2	0.1	0.0	0.1	5.2	0.0	2.8	0.0
dl1+win	**-5.2**	**-10.5**	**-6.8**	**-6.0**	**-4.2**	**-15.3**	**-0.2**	**-6.1**	**-4.3**	**-4.1**	**-27.6**	**-5.7**
dl1+bw	5.6	9.9	8.1	2.8	10.0	6.0	0.3	4.9	9.6	1.5	17.6	1.8
dl1+bmisp	**-10.8**	**-5.4**	**-4.9**	**-2.9**	**-7.0**	**-3.4**	**-2.4**	**-2.8**	**-7.6**	**-6.5**	**-0.2**	**-4.6**
dl1+dmiss	-0.7	-1.2	-0.4	-0.4	-1.4	-0.4	-0.5	-1.4	-0.2	-1.3	-1.8	-2.5
dl1+shortalu	**-4.1**	**-4.3**	**-1.0**	**-0.2**	**-1.6**	**-8.2**	**-0.1**	**-3.6**	**-1.4**	**-0.3**	**-4.0**	**-1.3**
dl1+longalu	-0.3	0.1	-0.3	0.1	-0.3	-0.4	0.0	-0.0	-0.7	0.0	-1.3	-0.3
dl1+imiss	0.0	0.0	0.8	0.1	0.3	0.0	0.0	0.0	1.0	0.0	0.4	0.0
Other	-2.5	15.4	21.4	-6.7	3.8	7.8	-18.8	0.5	6.3	-8.2	2.7	-5.9
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(a) CPI contribution breakdown (in percent) with four-cycle level-one cache.

Category	gap	gcc	gzip	mcf	parser
shalu	**37.0**	**13.1**	**39.2**	**3.3**	**38.2**
win	46.5	12.5	13.0	4.0	18.3
bw	1.6	7.1	4.4	0.4	2.4
bmisp	8.0	26.3	24.0	27.4	13.7
dmiss	17.4	26.8	8.6	82.1	28.8
dl1	4.9	10.9	17.0	4.5	9.2
imiss	0.4	2.0	0.1	0.0	0.0
lgalu	4.8	0.5	0.6	-0.0	0.1
shalu+win	**-26.8**	**-2.2**	**-9.1**	**0.1**	**-12.9**
shalu+bw	9.0	9.9	8.3	0.7	6.3
shalu+bmisp	**1.0**	**-5.7**	**-5.4**	**-2.3**	**-1.2**
shalu+dmiss	2.0	0.1	-1.2	0.4	-0.0
shalu+dl1	**0.4**	**-2.4**	**-7.8**	**-0.2**	**-3.2**
shalu+imiss	0.1	0.1	0.0	0.0	0.0
shalu+lgalu	-1.6	-0.4	-0.5	0.0	-0.0
Other	-4.7	1.4	8.8	-20.4	0.3
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(b) Breakdown with two-cycle issue-wakeup loop.

	gap	gcc	gzip	mcf	parser
bmisp	**11.7**	**25.5**	**27.8**	**26.7**	**16.8**
dl1	6.8	10.4	19.1	4.5	10.6
win	38.7	11.8	9.3	4.2	14.7
bw	3.8	12.8	8.0	0.5	4.0
dmiss	26.4	29.5	10.8	84.0	37.3
shalu	14.2	5.0	21.3	1.5	20.4
lgalu	6.0	0.3	0.8	0.0	0.1
imiss	0.8	2.5	0.1	0.0	0.1
bmisp+dl1	-1.7	-4.7	-2.4	-1.5	-1.8
bmisp+win	**2.1**	**9.6**	**12.4**	**5.3**	**14.2**
bmisp+bw	-1.2	-1.2	-2.6	-0.2	-1.3
bmisp+dmiss	**0.3**	**-1.3**	**-0.2**	**-16.4**	**-4.6**
bmisp+shalu	0.4	-3.0	-3.7	-1.1	-0.7
bmisp+lgalu	0.3	0.0	0.3	-0.0	0.0
bmisp+imiss	-0.2	-0.4	-0.0	-0.0	-0.0
Other	-8.4	3.2	-1.0	-7.5	-9.8
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(c) Breakdown with 15-cycle branch mispredict loop.

Table 4: **Breakdowns for optimizing a long pipeline.** Interaction costs 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); 'bmisp' → branch mispredictions; 'dmiss' → data-cache misses; 'shalu' → one-cycle integer operations; 'lgalu' → multi-cycle integer and floating-point operations; and 'imiss' → instruction cache misses. Due to space constraints, only a subset of the SPECint benchmarks are shown for **(b)** and **(c)**, but the benchmarks shown are representative of the suite. Note that 'Other', denoting the sum of all interaction costs not displayed, can be negative since the interaction costs can be negative. The machine modeled is described in Section 6.

make predictions as to the magnitude of the interactions.

The results of the analysis is shown in Table 4a (simulator parameters are in Table 6 in Section 6). For brevity, the breakdown presents only those interaction costs that involve data-cache accesses, labeled 'dl1' in the table. Notice first that data-cache accesses have a large cost, typically contributing 15–25% of the execution time.

We see that some of our hypotheses were correct: for instance, there are significant serial interactions between data-cache accesses and ALU operations (*dl1+shalu*), suggesting we could mitigate the long data-cache loop by reducing ALU latency (perhaps through value prediction [5, 19] or instruction reuse [30]).

We also notice, however, that the *magnitude* of the interaction varies significantly across benchmarks. This variability suggests that interaction costs could be useful in workload characterization: their magnitude gives a designer early insights into what optimizations would be most suitable for the most important workloads.

However, other conclusions from the analysis were not predicted beforehand. For example, it was hypothesized that data dependences with data-cache misses would cause a serial interaction with data-cache accesses. In reality, this interaction is very small: reducing data-cache misses is unlikely to mitigate the effect of the high latency data-cache loop.

We also see that the largest serial interaction for most benchmarks is with instruction window stalls. Thus, perhaps the most effective mitigation of the data-cache loop would be to increase the size of the instruction window — a result that may be difficult to predict before performing the analysis.

4.2 The issue-wakeup and branch mispredict loops

We also performed the same analysis for the issue-wakeup and branch misprediction loops. Due to space

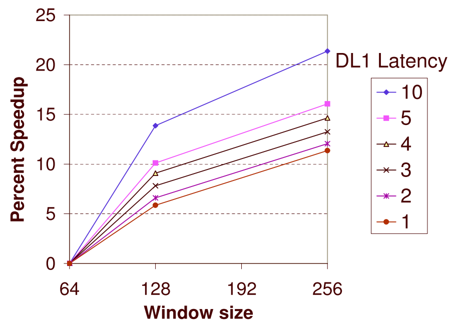


Figure 3: **Speedup from increasing window size for different level-one cache latencies.** As predicted from the negative interaction cost, increasing the window size has a larger benefit when level-one cache latencies are larger.

constraints, we will not present all of the data; instead, we only highlight the results of the analysis.

The issue-wakeup loop Suppose that a long pipeline demanded a two-cycle issue-wakeup latency, instead of the typical one. This will, of course, reduce performance, since ALU operations will not be able to issue back-to-back. Can we use serial interactions to determine how to mitigate the performance loss?

From the breakdown of Table 4b, we see significant serial interactions between ALU operations and several event classes: window stalls, branch mispredicts, and level-one cache accesses. The most significant interaction is, again, with window stalls; it is as large as -27% for *gap*. Because of this negative interaction, increasing the window size is more beneficial when the issue-wakeup latency is higher. For instance, we found that the speedup for *gap* when the window size is increased from 64 to 128 is 12% if the issue-wakeup latency is one and 18% if the latency is two, a difference of 50%.

The branch misprediction loop Finally, we consider the branch misprediction loop. Can we modify the microarchitecture to reduce branch misprediction costs? How about increasing the window size? Will that work to reduce branch misprediction loop cost in the same way it did for the other two loops?

The interaction costs in Table 4(c) reveal that the answer is no. Instead of a serial interaction, there is a *parallel* interaction between branch mispredictions and window stalls. This parallel interaction tells us there are a significant number of cycles that can be eliminated only by optimizing *both* classes of events simultaneously. In other words, reducing window stalls is not likely to significantly reduce branch misprediction costs.

For a couple of benchmarks, *mcf* and *parser*, we do see significant serial interactions with data cache misses (dmiss), however. Intuitively, this effect is likely due to cache-missing loads providing data that is used to determine a branch direction. Again, interaction costs help: we can quantify the importance of this effect for particular workloads, even determining the static instructions where it occurs, helping to guide prefetch optimizations.

4.3 Comparing with sensitivity study

A sensitivity study is an evaluation of one or more processor parameters made by varying the parameters over a range of values, usually through many simulations. Interaction costs can be viewed as a way to *interpret* the data obtained from a sensitivity study. Regardless of how they are computed, through multiple simulations or graph analysis, interaction costs explain *why* performance phenomena occur in a very *concise* way.

Let's explore this relationship by validating that the conclusions obtained from interaction-cost analysis and conventional sensitivity studies are the same. We perform the comparison by using a corollary of the serial interaction between the instruction window and load latency (the main result of Section 4.1). As the load latency becomes larger, increasing the size of the instruction window has increasing benefit. Since load latencies and window stalls occur in series with each other (because EP edges are in series with CD edges, as can be seen in Figure 2), increasing the latency of one will make both more dominant on the critical path[1].

Using this corollary, we performed the comparison by running several simulations to observe the speedup with increasing window size at different cache latencies (see Figure 3). Indeed, the interaction costs correctly predicted what the sensitivity study reveals: for instance, 50% greater speedup ((9-6)/6 x 100%) is obtained from increasing the window size from 64 to 128 when the data-cache latency is four instead of one.

From this example, we see the relationship between the two types of analyses. A full sensitivity study provides more information, *e.g.,* whether the curves in the plot are concave or convex; but interaction costs provide easier *interpretation* and concise *communication* of results. The interpretation is easy since the type and magnitude of the icosts have well defined meanings. The ease in communication comes from the ability to summarize a large quantity of data very succinctly. For example, the entire chart of Figure 3 can be summarized by simply stating that the two resources interact serially. Furthermore, due to the formulaic nature of interaction cost, the interpretation is available *automatically*, without the effort of a human analyst.

Summary. In this section, we showed that interaction costs can help microarchitects. When the the dependence graph is constructed by the simulator, architects can use interaction-cost-based breakdowns as a standard output of each simulation run. The overhead of building the graph during simulation in our research prototype is approximately two-fold slowdown, which we did not find overly burdensome. Using the same principles of sampling that facilitate the profiling solution of Section 5,

[1]We performed this same style of validation for the two analyses of Section 4.2 but do not present them due to space constraints.

Bit	When to set to '1'
1	Set to 1 if (1) *taken* branch or (2) load or store. Reset to 0 if L2 dcache miss.
2	Set to 1 if (1) L1 or L2 icache miss, (2) L1 or L2 dcache miss, or (3) tlb miss.

Table 5: **Description of signature bits.**

we found that the overhead could be reduced to approximately 10% without significantly impacting accuracy.

Perhaps even more exciting, however, is that all of this analysis can also be performed on real, deployed systems where resimulation and idealization is not an option. Hardware support for such analysis is the subject of the next section.

5 Measuring cost in hardware: Shotgun profiling

The challenge faced by hardware performance profilers is how to interpret their measurements, that is, how to translate the observed latencies and event counts into costs of bottlenecks (*e.g.,* if n cache misses occur, what percent of execution time should be blamed on cache misses?). Our profiler solves these problems by constructing fragments of our dependence graph that can be analyzed to compute interaction costs, just as if they were constructed in a simulator. Due to limited space, we describe the hardware algorithm without discussing detailed design tradeoffs.

The difficulty is that measuring detailed latency and dependence information for every dynamic instruction would require prohibitively expensive hardware. Our solution is to collect detailed information for only a sampling of instructions, *one* instruction at a time (similar to ProfileMe [9]). Later, post-mortem, the graphs of specific sequences of instructions are constructed by fitting these samples together, making use of *signature bits*. This approach of assembling a graph fragment from random samples is similar to the technique of shotgun genome sequencing [14], hence the name "shotgun" profiler.

Our solution works because, just as there are relatively few hot *control-flow* paths that comprise most of the execution, there are also relatively few *microexecution* paths, at the level of abstraction which affects the critical path. A microexecution path consists of control flow together with microarchitectural characteristics (*e.g.,* cache misses). In other words, we exploit a "locality of microexecutions," wherein the same microexecution paths recur many times during execution.

The profiler infrastructure consists of two components: a hardware performance monitor infrastructure and a post-mortem software graph construction algorithm. Each component will be discussed in turn.

5.1 Hardware Performance Monitors

If hardware expense was no concern, we could build graph fragments by collecting latency and dependence information for every dynamic instruction. Instead, we keep the hardware lightweight by collecting a relatively small amount of information that is used to construct the graph offline. We collect two types of samples:

- *Signature Sample.* A signature sample is *long* and *narrow*, consisting of two *signature bits* for each of the next 1000 dynamic instructions and a single "start" PC. Signature bits help identify a particular microexecution path and are set as shown in Table 5. The PC is of the first instruction that will appear in the graph (after a few instruction signature prefix, described below).
- *Detailed Sample.* A detailed sample is *short* and *wide*, consisting of latency and dependence information for a single dynamic instruction. Furthermore, a sequence of signature bits before and after the sampled instruction are collected. These will be used to "match" the detailed samples to appropriate segments of the signature trace. To minimize hardware costs, detailed samples are collected *sparsely* and for at most one dynamic instruction at a time.

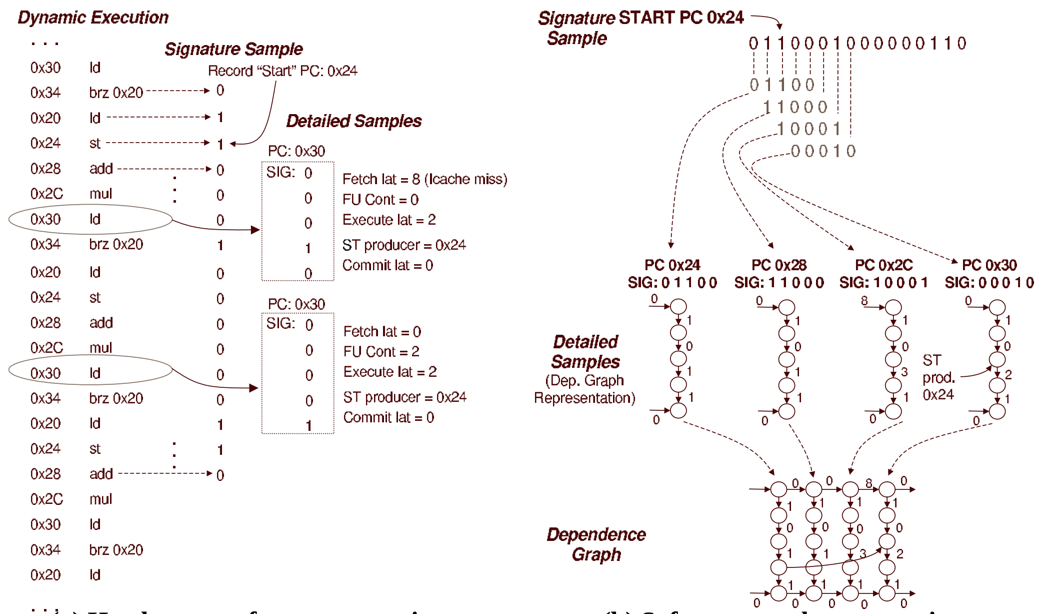
See Figure 4a for an illustration of the two types of samples. As each sample is taken, it is placed into a small on-chip buffer. When the buffer fills, an interrupt is raised and its contents are placed into a buffer in memory (or disk) for later (post-mortem) analysis.

Complexity. The hardware needed for collecting the *detailed sample* is similar to that proposed for the Alpha ProfileMe [9], and most of the requirements are similar to the support some current microprocessors already provide [7, 8]. The hardware for the *signature bits* is new, but the cost seems reasonable since (i) two bits is a small amount of information to maintain and (ii) they typically indicate a processor stall, which makes setting them unlikely to be on a time-critical circuit path.

5.2 The Software Graph Construction Algorithm

After samples have been collected via the hardware performance monitors, software uses the information to construct dependence graph fragments, which can then be analyzed as if they were constructed in a simulator. This offline analysis is relatively efficient since we do not need to analyze the entire graph but only a relatively small number of graph fragments.

The algorithm works by first selecting a *signature sample* at random, which serves as a "skeleton for the graph to be built. (The random selection ensures each signature sample is chosen with equal probability, which naturally gives priority to hot microexecution paths.) The goal of the algorithm is to fill in this skeleton with *detailed samples* to form a latency-labeled dependence graph. To accomplish this, an appropriate detailed sample is placed into the graph for each dynamic instruction in the trace, where "appropriateness" is determined by the PC and the signature bits.



(a) Hardware performance monitors **(b) Software graph construction**

Figure 4: **The profiler infrastructure consists of two parts. (a) Hardware performance monitors.** Our hardware performance monitors collect two types of samples: signature samples and detailed samples. For illustration, the figure shows one signature bit per instruction and collection of the bits for two instructions before and after each detailed sample. For greater accuracy, our design uses two signature bits per instruction (see Table 5) and collects signature bits for ten instructions before and after each detailed sample (see Figure 5a). **(b) Post-mortem software graph construction.** The dependence graph is constructed by concatenating detailed samples, so that the resulting graph is representative of the microexecution denoted by the signature sample.

For example, consider building the graph nodes for the first instruction in the signature sample of Figure 4. The first instruction has PC of $0x24$, so we look up detailed samples with this PC. Then, we select the detailed sample whose signature bits (most closely) matches the corresponding bits in the signature sample. (If no detailed sample is found for the PC, which empirically happens less than 2% of the time, we infer everything possible from the binary and use default values for the unknown latencies.) Finally, the nodes for this instruction are constructed from the selected detailed sample.

Remember that a signature sample consists solely of a start PC and the signature bits, *i.e.,* to reduce hardware costs the PCs of other instructions are not recorded. Thus, we need to use some intelligence to infer the PC of each dynamic instruction in the signature sample. For direct conditional branches, we include the branch direction in the signature bits and lookup the binary for the target address. For indirect branches, we include the branch target address in the detailed samples. The details are described in the complete algorithm for constructing a graph fragment in Figure 5a.

Note that some of the detailed information required to build the graph does not need to be collected dynamically from hardware. Instead, it can be inferred statically from the program binary and the machine parameters (*e.g.,* pipeline length). See Figure 5b for a listing of how various dependences and latencies are collected.

6 Validating profiler accuracy

In this section, we measure the accuracy of our hardware profiler described in the previous section. We evaluate its accuracy by comparing the breakdowns it produces with the more accurate breakdowns produced (i) from full dependence graphs constructed in a simulator; and (ii) from running multiple idealized simulations.

We find that the profiler's accuracy is (on average) within 9% of the full dependence graph analysis, and within 11% of multiple simulations. The first error is due to sampling and the (intended) simplicity of the signature used in the profiler. The difference between the 9% and 11% error is due to approximations in the dependence graph (again, this is intended, for the sake of graph complexity).

Methodology. We simulate the out-of-order processor described in Figure 6, using the SPEC2000int suite (as optimized Alpha binaries) with reference inputs. Our simulator is built upon the SimpleScalar tool set [4]. We skipped eight billion dynamic instructions and then performed detailed timing simulation for 100 million.

We use the multiple-simulation approach as our baseline. There is one simulation for each category in the breakdown where the simulation *idealizes* the appropriate set of event classes (see Table 1 in Section 2 for examples of idealizations). Table 7 shows breakdowns computed three ways for the same categories and machine configuration used in Table 4a. For the graph anal-

```
1.     Randomly select a *signature sample* for the skeleton.
       Call the starting PC in this sample the $StartPC$.
2.     For each instruction $i$ from $StartPC$ to end of fragment
2a.      Get from database all detailed samples with $i$'s PC.
2b.      Select the detailed sample whose signature bits most closely
         matches the portion of the signature sample $10$ instruction
         before $i$ to $10$ instructions after. The closeness of a match is
         judged by the number of identical bits.
2c.      Append sample's nodes and edges to the graph (see Fig. 4).
2d.      Determine PC of next instruction, $i+1$ (call PC of $i$ $CurPC$
         and PC of $i+1$ $NextPC$):
2d1.       If $i$ is not a branch, $NextPC \leftarrow CurPC + 4$
2d2.       If $i$ is a direct branch and signature bit 1 of $i$ is 1,
             Compute branch target and set $NextPC$ equal to it
           Else $NextPC \leftarrow CurPC + 4$
2d3.       If $i$ is a call, push target PC onto stack
           For returns, pop stack (if nonempty) and set $NextPC$ to
           that PC
2d4.       If $i$ is an indirect branch, set $NextPC$ equal to target PC in
           detailed sample for $i$
2e.      Check for inconsistency (see caption).
```

(a) Algorithm for constructing a graph fragment in software

dep	col	latencies	col
DD	S	icache misses, itlb misses	D
FBW	S	constant latency (1 cycle)	S
CD	S	constant latency (0 cycle)	S
PD	D	branch recovery latency	S
DR	S	constant pipeline latency	S
PR	reg: S, mem: D	constant latency (0 cycle)	S
RE	S	functional unit contention	D
EP	S	Execution latency	D
PP	D	constant latency (0 cycle)	S
PC	S	constant pipeline latency	S
CC	S	store BW contention	D
CBW	S	constant latency (1 cycle)	S

(b) How dependences and latencies are collected

Figure 5: **Graph-construction algorithm and how latencies and dependences are collected. (a)** Note that using the target address in the detailed sample sometimes leads down a control path that is inconsistent with the signature sample (it is consistent 60–99% of the time). In these cases, we attempt to detect the inconsistency by looking for impossible signature bit settings. For instance, if an instruction on the signature sample has its first bit set to 1, it should be a load, store, or branch. If the PC computed by the algorithm does not correspond to one of these instruction types in the program binary, we know there is an inconsistency and abort building that graph segment (since analyzing such a graph would lead to error in the results). We have found that 95-100% of the errant graphs are indeed discarded using this technique. **(b)** 'D' means the dependence or latency is collected dynamically; 'S' stands for statically. Dependences and latencies that must be determined dynamically are measured in hardware (in the *detailed samples*). Those that can be determined statically are inferred from the program binary or the machine description. Besides the dynamic dependence and latency information, the target PC of indirect branches is also recorded in the detailed sample.

ysis in a simulator (*fullgraph*) and the profiler (*profiler*) results are shown as absolute error relative to multiple simulations (*multisim*).

Discussion. From the breakdown tables, we make two observations. First, the profiler tracks the dependence-graph analysis very closely, with average error of 9%. Thus the approximations that lead to inexpensive hardware profiling (*e.g.,* sampling and incomplete latency and dependence information) represent a good accuracy versus complexity tradeoff.

Second, the profiler also tracks multiple simulations closely, with an average error of 11%. Thus, our dependence-graph model (described in Section 3) is a reasonable approximation of the simulated processor.

7 Related Work

Previous work into microarchitectural performance analysis takes on many forms. Event counters and utilization metrics [1,39] have become standard and, before out-of-order processors, was all that was needed. When instructions are executed in parallel, however, simply counting events is not enough to know their effect on execution time. In response to the problems with counters, ProfileMe [9] supports pair-wise sampling, where the latencies and events of two simultaneously in-flight instructions are recorded. With these pair-wise samples, one can determine the degree to which two instructions' latencies overlap in time. Also, the Pentium 4 [8,32] has a limited ability to account for overlapping cache misses. These performance monitoring facilities do not appear amenable to computing a complete breakdown of execution time, however. We introduce interaction cost to provide this level of interpretability.

There are several works that aim to interpret the parallelism of out-of-order processors through fetch [10,21] and commit attribution [16, 20, 22, 24, 25, 35], and at least one that combines attribution with some dependence information [26]. In these approaches, specific instructions and events are assigned blame for wasted fetch bandwidth or commit bandwidth, respectively. We have found these analyses do, indeed, accurately compute the cost of certain classes of events, which was their intended purpose. They have not been used to compute interaction costs, however.

Several researchers have explored criticality and slack, two useful metrics for exploiting the parallelism in out-of-order processors [6,11–13,23,27–29,33,34,36, 37]. Our notion of interaction cost extends these works by answering questions about nearly-critical paths, such as (i) "Which critical dependences are most important to optimize?" and (ii) "Which nearly critical dependences should I optimize along with the critical ones?"

One of the above papers, by Tune et al. [37], was the

Dynamically Scheduled Core	64-entry instruction window, 6-way issue, 15-cycle pipeline, perfect memory disambiguation, fetch stops at second taken branch in a cycle.
Branch Prediction	Combined bimodal (8k entry)/gshare (8k entry) predictor with an 8k meta predictor, 4K entry 2-way associative BTB, 64-entry return address stack.
Memory System	32KB 2-way associative L1 instruction and data (2 cycle latency) caches, shared 1 MB 4-way associative 12-cycle latency L2 cache, 100-cycle memory latency, 128-entry DTLB; 64-entry ITLB, 30-cycle TLB miss handling latency.
Functional Units (latency)	6 Integer ALUs (1), 2 Integer MULT (3). 4 Floating ALU (2), 2 Floating MULT/DIV (4/12), 3 LD/ST ports (2).

Table 6: **Configuration of simulated processor.**

	gcc			parser			twolf		
	multisim	fullgraph	profiler	multisim	fullgraph	profiler	multisim	fullgraph	profiler
dl1	16.1	+2.2	+2.5	17.0	+2.0	+2.4	17.1	+2.4	+2.9
win	11.7	+1.9	-1.2	15.0	+2.3	-3.2	22.2	+2.9	-1.7
bw	10.8	-2.6	-1.4	3.5	-0.7	-0.4	4.4	-0.6	-0.2
bmisp	26.8	-0.5	-2.8	17.3	-0.8	-0.8	24.3	-0.2	-0.3
dmiss	25.3	+0.9	+2.5	32.5	+0.4	+0.6	34.2	+0.2	-0.6
shalu	4.7	+0.4	+0.8	18.3	+1.4	+2.8	8.0	-0.2	+1.6
lgalu	0.3	+0.0	+0.0	0.1	-0.0	+0.0	4.3	-0.1	+0.6
imiss	2.1	+0.0	-1.4	0.1	-0.0	-0.1	0.1	-0.0	-0.1
dl1+win	-3.4	-0.8	-0.5	-5.1	-0.9	-0.1	-3.2	-0.9	-0.9
dl1+bw	10.4	-0.4	-1.1	5.7	-0.8	-1.6	1.8	-0.3	-0.7
dl1+bmisp	-7.4	+0.3	+0.6	-2.2	-0.6	-0.8	-5.6	-0.9	-1.1
dl1+dmiss	-1.2	-0.2	-0.2	-1.3	-0.0	+0.1	-0.4	-0.9	-0.6
dl1+shalu	-1.5	-0.2	-0.7	-4.5	+0.9	-0.3	-0.8	+0.5	-0.3
dl1+lgalu	-0.3	-0.0	+0.0	-0.0	+0.0	-0.0	-0.1	+0.1	-0.0
dl1+imiss	0.4	-0.1	-0.1	-0.0	+0.0	+0.0	-0.0	+0.0	+0.0

Table 7: **Measuring accuracy of profiler.** Validation was performed on the same CPI contribution breakdown and machine model as in Table 4a (with results expressed in percent of the total CPI). Due to space constraints only three benchmarks are shown, but they are representative of the rest of SPECint2000. For the *fullgraph* and *profiler* columns, the absolute error relative to *multisim* is reported. The percent error per category between the profiler and the full dependence graph is computed as $abs(profiler - fullgraph)/(multisim + fullgraph)$, and the averages (excluding categories under 5%) are: 10% for *gcc*, 8% for *parser*, 9% for *twolf*. The average error per category between the profiler and multiple simulations is computed as $abs(profiler)/multisim$, and the averages are: 12% for *gcc*, 14% for *parser*, 9% for *twolf*. Overall, for the twelve SPECint2000 benchmarks, the average error between the profiler and (i) the dependence graph is 9% (ii) multiple simulations is 11%.

first to use the dependence graph to compute the cost of *individual* instructions in a simulator (we employ their algorithm). The focus of our paper is on how the costs of not only instructions but also machine resources *interact* in an out-of-order processor. We also provide a design for a hardware profiler, so that the analysis can be performed on real systems.

The MACS model of Boyd and Davidson [3] assigns blame for performance problems to one of four factors: the machine, application, compiler-generated code, or compiler scheduling. They accomplish this by idealizing one factor at a time (to determine its cost). In comparison to this work, we focus only on fine-grain microarchitectural events (as opposed to compiler decisions) and introduce a methodology for measuring interactions.

Yi, et al. [38] use a Plackett and Burman design to reduce the number of simulations required in a sensitivity study. However, their work does not quantify and interpret specific interactions between events. Standard allocation and analysis of variance (ANOVA) techniques do, in fact, quantify these interactions [18]. ANOVA is inadequate for our purposes, however, for two reasons: (1) squaring of effects reduces their interpretability and (2) no distinction is made between positive and negative (parallel and serial) interactions.

8 Conclusion

The primary contribution of our work is establishing *interaction cost* as a methodology for bottleneck analysis in complex, modern microarchitectures. Interaction cost permits one to account for all cycles of execution time, even in an out-of-order processor, where instructions are processed in parallel.

We have also provided a relatively inexpensive hardware profiler design (close to the complexity of ProfileMe [9]) that enables measuring interaction cost in real systems. With this technology, not only microarchitects, but also software engineers, compilers and dynamic optimizers can make use of the deeper understanding of performance bottlenecks.

For instance, feedback-directed compilers could favor prefetching cache misses that serially interact with branch mispredicts. Performance-conscious software engineers could identify the most important procedures and instructions for optimization and determine why the performance problems exist. Dynamic optimizers could save power by intelligently reconfiguring hardware structures. Finally, real workloads could be analyzed on real hardware, such as large web servers running a database.

Acknowledgements. We thank Mary Vernon, David Wood, and Amir Roth for contributions to this work. We also thank Sarita Adve, Bradford Beckmann, Mark Buxton, Jarrod Lewis, David Mandelin, Milo Martin, Anat Shemer, Dan Sorin, Manu Sridharan, Renju Thomas, Min Xu, and the anonymous reviewers for comments on drafts of this paper. Finally, we thank the Wisconsin Architecture affiliates for feedback on early presentations of this work. This work was supported in part by National Science Foundation grants (CCR–0326577, CCR–0324878, CCR–0225610, EIA–0205286, CCR–0105721, EIA–0103670, EIA–9971256, and CDA–9623632), an NSF CAREER award (CCR–0093275), IBM Faculty Partnership Award, a Wisconsin Romnes Fellowship, and donations from Intel, Microsoft, and Sun Microsystems. Hill's sabbatical is partially supported by the Spanish Secretaría de Estado de Educución y Universidades. Fields was partially supported by NSF Graduate Research and Intel Foundation Fellowships.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, Nov 1997.

[2] E. Borch, E. Tune, B. Manne, and J. Emer. Loose loops sink chips. In 8th *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[3] E. L. Boyd and E. S. Davidson. Hierarchical performance modeling with MACS: A case study of the Convex C-240. In 20th *International Symposium on Computer Architecture*, May 1993.

[4] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, Jun 1997.

[5] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In 26th *International Symposium on Computer Architecture*, May 1999.

[6] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *Kool Chips Workshop in conjunction with MICRO 33*, Dec 2000.

[7] Intel Corporation. Intel Itanium 2 processor reference manual for software development and optimization. Apr 2003.

[8] Intel Corporation. Intel Pentium 4 processor manual. In *[http://developer.intel.com/design/pentium4/manuals/]*, 2003.

[9] 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 30th *International Symposium on Microarchitecture*, Dec 1997.

[10] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In 34th *International Symposium on Microarchitecture*, Dec 2001.

[11] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In 29th *International Symposium on Computer Architecture*, May 2002.

[12] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In 28th *International Symposium on Computer Architecture*, Jun 2001.

[13] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. Oct 1999.

[14] R. D. Fleischmann et al. Whole-genome random sequencing and assembly of haemophilus-influenzae. *Science*, 269:496–512, 1995.

[15] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In 29th *International Symposium on Computer Architecture*, 2002.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 3rd edition, 2002.

[17] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In 29th *International Symposium on Computer Architecture*, 2002.

[18] Raj Jain. *The Art of Cumpter Systems Performance Analysis*. Wiley Professional Computing, 1991.

[19] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In 29th *International Symposium on Microarchitecture*, Dec 1996.

[20] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In 3rd *International Symposium on High Performance Computer Architecture*, Feb 1997.

[21] S. Patel, M. Evers, and Y. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In 25th *International Symposium on Computer Architecture*, Jun 1998.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In 34th *International Symposium on Microarchitecture*, December 2001.

[23] R. Rakvic, B. Black, D. Limaye, and J. P. Shen. Non-vital loads. In 8th *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[24] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. Oct 1998.

[25] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In 15th *Symposium on Operating Systems Principles*, Dec 1995.

[26] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In 10th *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[27] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In 8th *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[28] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In 34th *International Symposium on Microarchitecture*, Dec 2001.

[29] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In 34th *International Symposium on Microarchitecture*, Dec 2001.

[30] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In 24th *International Symposium on Computer Architecture*, 1997.

[31] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In 29th *International Symposium on Computer Architecture*, 2002.

[32] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, Jul 2002.

[33] S. T. Srinivasan, R. Dz ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In 28th *International Symposium on Computer Architecture*, Jun 2001.

[34] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In 31st *International Symposium on Microarchitecture*, Nov 1998.

[35] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In 27th *International Symposium on Computer Architecture*, Jun 2000.

[36] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In 7th *International Symposium on High-Performance Computer Architecture*, Jan 2001.

[37] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In 11th *International Conference on Parallel Architectures and Compilation Techniques*, Sep 2002.

[38] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In 9th *International Symposium on High Performance Computer Architecture*, Feb 2003.

[39] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing '96*, 1996.