Dependence-Based Scheduling Revisited: A Tale of Two Baselines

Pierre Salverda Craig Zilles

Department of Computer Science,
University of Illinois at Urbana-Champaign
{salverda, zilles}@uiuc.edu

Abstract

Previous work proposes a dependence-based scheduling technique in which the scheduling window is partitioned into a number of FIFO buffers, the heads of which constitute the set of instructions eligible for execution each cycle. In that work, empirical evaluation promises very modest IPC losses relative to a conventional scheduler, but we find in our own implementation that losses are severe — about six-fold higher than those originally reported. We show that those losses arise because the dependence-based steering policy, which is responsible for distributing instructions among the FIFO buffers, must frequently stall for lack of a suitable buffer into which the next instruction can be slotted. That problem, in turn, arises because dynamic dataflow is simply not of a shape that is amenable to being mapped onto just a few FIFO buffers, at least not by means of the proposed steering policy. We postulate that the original study produced good results because it evaluated the new scheduler in a machine whose performance bottleneck lay outside the execution core. The problems introduced by the steering logic were, as a result, hidden.

1 Introduction

With tight constraints on circuit complexity, area and power, one of the most problematic hardware components in out-of-order superscalar designs is the dynamic scheduler. It is key to the machine's ability to sustain high performance in terms of instructions per clock (IPC), but it is at the same time one of the largest, most complex and power-hungry parts. Underlying both its performance pros and its implementation cons is the fact that the scheduler provides, within the scope afforded by the instruction window, a *dataflow execution model*. That is, instruction execution is generally constrained by dataflow dependences only; the machine's ability to find and exploit instruction-level parallelism (ILP) is limited primarily by the window size. But this capability comes at the cost of a complex, broadcast-based scheduling technique whose long wires and numerous comparators pose some challenging problems for meeting a design's power, area and cycle time goals. And expanding the window size to extract more ILP merely exacerbates those problems.

One way to tackle this is to restrict the extent to which the scheduler supports the dataflow execution model. The *dependence-based scheduler* proposed by Palacharla *et al.* [7] does just that. It partitions the issue queue into a number of FIFO buffers, permitting only instructions at the heads of each to be eligible for issue each cycle. This constitutes a restriction on the execution model in the sense that it is now more than just dataflow dependences that constrain instruction execution: there is now also an *in-order* constraint imposed by each FIFO buffer. Put another way, out-of-order execution can now arise only through "slip" among the buffers. Scheduling logic benefits from such an organization because it need monitor the readiness of, and select from among, only the heads of each FIFO buffer; the complex broadcast-based circuitry is entirely eliminated.

Palacharla *et al.* showed that their dependence-based scheduler is capable of delivering IPC which rivals that of a conventional machine, with average loss being about 5%. Such good results are important, not only because they imply that implementation benefits need not compromise IPC, but also because they confirm that a restricted execution model is an effective means for extracting ILP. The latter result embeds a rather fundamental statement about dynamic dataflow. This is because the dependence-based scheme exploits properties of dataflow to overcome its restricted execution model. Specifically, it uses dataflow dependences to control the allocation (steering) of instructions to FIFO buffers, slotting consumers directly behind the producers of their operands. As a result, all instructions are dataflow dependent on those ahead of them in their respective FIFO. This simple invariant ensures that the in-order issue constraint is always *subsumed* by dataflow constraints. But this will only be effective to the extent that the instruction stream embeds dataflow that lends itself to being mapped onto the FIFO buffers. That good IPC has been demonstrated means, therefore, that dataflow is indeed amenable to being decomposed into a few chains of dependent instructions, the independent execution of which suffices for extracting ILP.

However, we believe this to be wrong. In our own evaluation of the dependence-based scheduler, we obtain performance losses about sixfold higher than those in the original study. Taken by themselves, our results paint a very different picture about dataflow and the efficacy of the dependence-based machine's restricted execution model. Such discrepancies prompt the detailed investigation we report on here.

We begin in Section 3 with an exploration of the basic factors at work in a dependence-based machine. We use critical path analysis to show that the machine suffers from a preponderance of *fetch-criticality*, meaning it is operating in a regime in which performance is determined mainly by the rate at which instructions are being delivered into the window. This is caused by the dependence-based steering rules, which frequently stall instruction dispatch for lack of a suitable slot into which the next instruction can be placed. This problem arises, ultimately, because of basic properties of dynamic dataflow. Simply put, dataflow does not readily decompose into just a few dependence chains, but rather into a large number of chains, most of them very short. The net effect is a "shallow and wide" distribution of instructions across the FIFO buffers, and hence a reduction in the issue queue's effective capacity. Thus is lost one of the main benefits of out-of-order execution — that of *buffering* instructions in order to *decouple* dispatch from the order and rate at which instructions can be executed. Without this capability, the machine is restricted in three ways: (1) there is a reduction in exploitation of ILP found across loop iterations; (2) there is limited ability to progress forward in the instruction stream to reach important instructions that lie ahead; and (3) there is a reduction in the machine's ability to find and exploit memory-level parallelism (MLP).

With a proper understanding of the machine's behavior in hand, we can then offer explanations for the good performance results reported in the original study. In Section 4, we postulate that the baseline machine into which the dependence-based scheduler was introduced, and against which it was evaluated, was seldom stressing the execution core. Its critical path was most likely dominated by fetch-criticality and by recovery from branch mispredictions. In short, the new scheduler introduced its fetch-criticality problem *in the shadow of* a pre-existing instruction supply constraint. Its deleterious effect on performance was thus hidden. In our own evaluation of the machine, the underlying instruction supply constraint is less severe, so the scheduler's problems come to the fore. This does not imply the original study's choice of baseline microarchitectural parameters were wrong, but it does mean the efficacy of the dependence-based scheduler was never really tested.

We conclude in Section 5 with a summary of our main results. There, we also comment on some of our methodological techniques, which we feel can be useful in a broader context than this work alone. First, however, we briefly review the dependence-based microarchitecture and present our own evaluation of it, the results of which motivated this study.

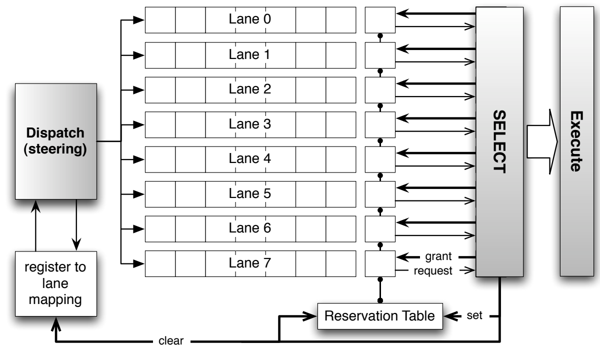


Figure 1: The dependence-based scheduler. The diagram shows a partitioning of a 64-entry issue queue into 8 lanes (FIFO buffers), each 8-deep. The head (oldest) entry in each lane is conceptually distinct from its successors since only it is a candidate for selection. Operand availability is communicated to all head entries via the reservation table, whose 1-bit entries are set when the producer of the corresponding value is selected, and cleared when that value becomes available. Steering logic records the lane to which the producer of each architected value is sent; scheduling logic clears this record (if it is still current) when that producer is issued.

2 Background and motivation

Modern superscalar machines support a dataflow-oriented execution model by means of a complex, broadcast-based scheduling technique. This involves two mutually dependent operations. In the first, called *select*, instructions whose operands are ready, and for which execution resources are available, are issued to the functional units. The second, called *wakeup*, involves notifying all instructions in the issue queue of the imminent availability of values to be produced by the selected instructions. Consumers of those values thus become eligible for selection in subsequent cycles. In their 1997 study, Palacharla *et al.* [7] found that scheduler latency — wakeup delay, in particular — dominates that of other microarchitectural components. And since the need for back-to-back issue of dependent instructions precludes pipelining its operation, the scheduler imposes a lower bound on the machine's achievable clock period.

2.1 Dependence-based scheduling

The dependence-based scheduler was proposed as a means for facilitating a faster clock. Figure 1 shows its design. The issue queue is partitioned into a number of FIFO buffers (we will henceforth call them *lanes*), each of which holds a subset of all dispatched instructions. Dispatch logic *steers* instructions into the tail-end of the various lanes, using

Instruction supply	perfect instruction cache. perfect unconditional branch prediction; modest gshare conditional branch predictor.
Front-end	8-wide, unknown depth.
Window	128-entry ROB. 64-entry unified issue queue.
Execute	8 universal units, 4 memory ports. all instructions unit latency. loads wait for older store addresses.
Memory	L1: 32KB, 2-way, 1-cycle hit. L2: perfect, 6-cycle hit.
Back-end	16-wide.

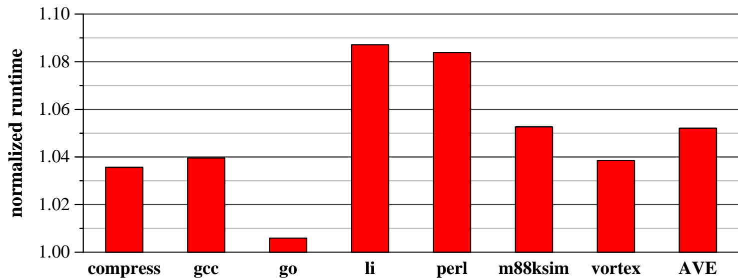


Figure 2: The original evaluation of dependence-based scheduling. The table on the left shows the simulator parameters for the baseline machine against which the dependence-based machine was evaluated by Palacharla *et al.* The graph on the right shows the runtime of the latter relative to that of the former. Data is interpolated from the absolute IPC numbers originally published [7].

dataflow dependences among them to make its decisions. The heuristic used for this purpose proceeds as follows.

1. Pick a non-full lane whose tail entry holds the producer of one of the dispatching instruction's operands.

2. If no such lane can be found, pick an empty lane.

3. If no empty lane is available, stall.

Thus, an instruction is slotted into a non-empty lane if and only if (one of) its producer(s) immediately precedes it there. This will not be possible when all producers have issued or none of them reside at the tail of a non-full lane. In such cases, an empty lane is sought. If that too fails, dispatch is stalled.

Because all instructions within a lane are dependent on those ahead of them, only the heads of each lane are candidates for issue. The select logic therefore need concern itself with only a small subset of the issue queue (8 out of 64 entries in Figure 1). For the same reason, back-to-back issue can be achieved by advertising result availability to only the heads of each lane. These simplifications have enormous implementation benefits. Most importantly, elimination of the long wires and all the comparators needed by the broadcast-based wakeup logic reduces scheduler latency enough to remove it from the processor's timing critical path. The same simplification also offers savings in terms of both area and power. Of course, all these benefits must be weighed against the potential for IPC loss.

2.2 Performance

The dependence-based scheduler exploits properties of dataflow to overcome the constraints imposed by its restricted execution model. Specifically, because the steering logic permits instructions to collocate at a lane only if they are part of a dataflow dependence chain, it is guaranteed that the in-order constraint imposed by that lane is always subsumed by dataflow constraints. In other words, dataflow dependences render the in-order constraint benign — but only so long as a program's instruction trace embeds dataflow that is amenable to being mapped onto the lanes by means of the steering rules described above. It is not immediately obvious to what extent this ought to be possible: it depends on the shape of, and distribution of ILP within, the dynamic dataflow graph. Neither of these properties is easily characterized.

Palacharla *et al.* render the whole question moot with an empirical evaluation of their scheme. Figure 2 reproduces their results. An 8-wide baseline machine is compared to a dependence-based counterpart that partitions the 64-entry issue queue into 8 lanes, each 8-deep. Instruction dispatch is modified to steer instructions into the lanes using the heuristic of Section 2.1. As the graph in Figure 2 shows, the dependence-based machine experiences very modest slowdowns: about 5% on average, never worse than 10%.

These results are significant because they confirm that all of the aforementioned implementation benefits can be won at modest cost to IPC. More importantly, the ability to sustain good IPC *in spite of* a restricted out-of-order execution model has two important implications. First, the flexibility afforded by the less constrained dataflow execution model appears to be superfluous; it is possible to exploit dataflow properties to build a simple, complexity-effective scheduler. Second, dataflow has the propitious quality that it can be quite readily decomposed into (just a few) chains of dependent instructions, the parallel and independent execution of which suffices for extracting ILP. The latter is an especially important result because it has broad relevance, particularly for other dependence-based schemes developed subsequent to the original study [3, 4].

These compelling results stand in stark contrast to those we obtained. We incorporated a version of the dependence-

Instruction supply	perfect instruction cache. perfect unconditional branch prediction; aggressive tournament conditional branch predictor.
Front-end	8-wide, 10 stages to dispatch.
Window	128-entry ROB. 64-entry unified issue queue.
Execution	8 universal units, 4 memory ports. latencies similar to Alpha 21264. ideal memory disambiguation.
Memory	L1: 32KB, 2-way, 2-cycle hit. L2: perfect, 12-cycle hit.
Back-end	8-wide.

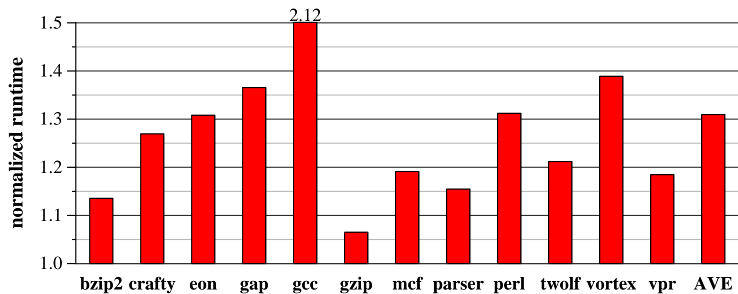


Figure 3: Our own evaluation of dependence-based scheduling. Like Figure 2, the table on the left shows the baseline machine configuration. As before, the dependence-based machine partitions the monolithic issue queue into 8 lanes, each 8-deep. The graph on the right plots its runtime relative to that of the baseline. Note how the scale on the y-axis differs from that in Figure 2.

based scheduler into our own simulator infrastructure, doing so as part of a more general analysis of the repercussions of the above results. Given the broader context of that analysis, it was *not* our aim to replicate the original numbers. We therefore evaluated a different benchmark suite (SPEC 2000 instead of SPEC 95) and our simulator parameters differ somewhat from those used in the 1997 study. However, we share with Palacharla *et al.* the overall objective of modeling, not a specific state-of-the-art microarchitecture, but one in which the execution core is exercised to the fullest. Like them, we assume an aggressive instruction-supply, with ideal instruction cache and perfect prediction of unconditional control flow. But we retain imperfect conditional branch prediction because the effect of mispredictions can be important for scheduling. Likewise, we model an aggressive, but imperfect, data-supply to capture the effects of variable latency operations.

Our baseline machine model and simulation results appear in Figure 3. Instead of modest penalties in the region of 5%, we find slowdowns are, on average, sixfold higher at about 30% and, in the majority of cases, are at or above 20%. The case of `gcc`, which is a clear outlier at more than double the runtime, also hints at a lack of performance robustness in the dependence-based scheme. To put these performance losses in perspective, it is worth noting that they are not very different from what our monolithic machine achieves when equipped with a 16-entry issue queue. Equally, a scheduler that pipelines its wakeup-select logic has been shown to suffer very similar slowdowns [8].

While our results might render the dependence-based scheduler less appealing from a complexity-effective point of view, it is perhaps tempting to dismiss them as an artifact of a different experimental framework. Of course, it is ultimately differences between the two simulator infrastructures that are to blame for the discrepancies, but such a coarse diagnosis is unsatisfactory — for two reasons. First, it does not explain why trends in *relative* performance are so different; one would expect simulator vagaries to be largely accounted for by the comparative evaluations. Second, it does not answer the more fundamental question about the potential for exploiting basic properties of dataflow to build a scheduler based on the independent execution of dependence chains.

3 Deconstructing scheduler behavior

In this section, we analyze the behavior of (our implementation of) the dependence-based scheduler in detail. We show, first, that it suffers from an instruction supply problem and, second, that it is an interaction between the steering rules and basic properties of dynamic dataflow that are to blame. Underpinning both of these results is *critical path analysis*, a technique we use to *quantitatively* characterize the behavior of a machine. Although aggregate statistics might be used to arrive at some of the conclusions we do, a critical path analysis is the most direct and accurate means for doing so. It permits us to zoom directly in on those aspects of behavior that are truly having an effect on performance. In this regard, aggregate statistics can be very misleading because they do not always correlate with observed IPC.

3.1 Where the cycles went

We use the dependence graph model developed by Fields *et al.* [2] to find the critical path through a program execution. That model defines three events for each instruction: dispatch into the window, execution, and retirement from the back-end. For each of these events, a dynamic instruction contributes a node to a dependence graph for the entire program. Edges among those nodes capture the microarchitectural and dataflow constraints that together determine a program's behavior. We delineate the critical path during a postmortem analysis of our simulator's execution

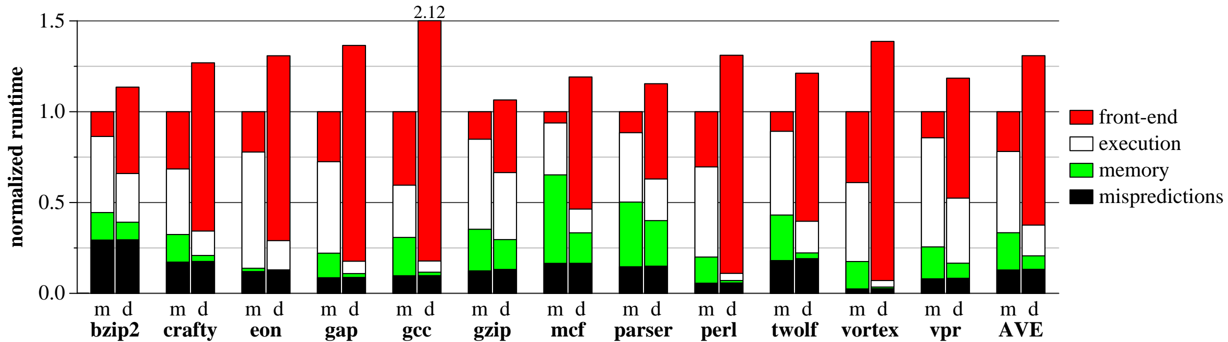


Figure 4: Critical path breakdowns. Each pair of bars shows the critical path breakdown for the monolithic baseline ('m') and the dependence-based ('d') machines. Cycle counts are normalized to the former.

trace. Moving backwards from younger to older instructions, we identify a (not necessarily unique) sequence of the aforementioned events that are responsible for the observed runtime. Using the critical path thus identified, we then attribute cycles to various aspects of behavior.

Figure 4 shows the resulting critical path breakdown for our baseline and dependence-based machines. A clear difference between the two immediately stands out. In the dependence-based machine, about 65% of runtime is spent in the machine's front-end, meaning performance is determined mainly by the rate at which instructions are delivered into the window. The monolithic baseline machine, in contrast, spends less than 20% of its time waiting for instructions, devoting most time instead to instruction execution and cache misses (the 'execution' and 'memory' categories). These two very different modes of operation are referred to as *fetch-* and *execute-critical*, respectively [2].

A fetch-critical regime is indicative either of insufficient front-end bandwidth (relative to back-end execution bandwidth) or of back pressure in the pipeline preventing instructions from dispatching into the window. The former is certainly not the problem given that both the monolithic and dependence-based machines share the same aggressive front-end architecture. So we can turn to the back-pressure problem. This usually arises when a resource constraint in the execution core — a full issue queue, for example — prevents instructions from being dispatched. However, as the data in Figure 5 shows, ROB and issue queue occupancy in the dependence-based machine are both at 30% for more than 75% of the time; both structures average about 25% occupancy.[1] Clearly, a window buffering constraint is not the problem. But Figure 5 also shows that occupancy of the lane heads is almost always 100%. That is, although aggregate lane occupancy is low, all lanes tend to have at least one

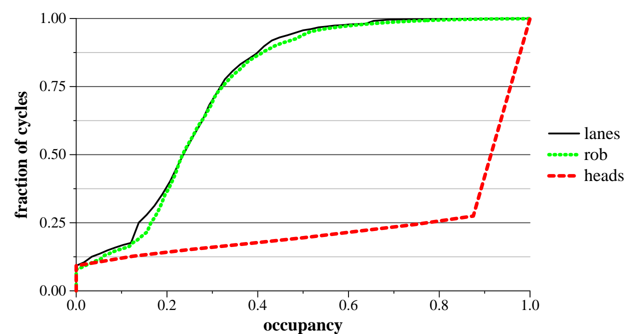


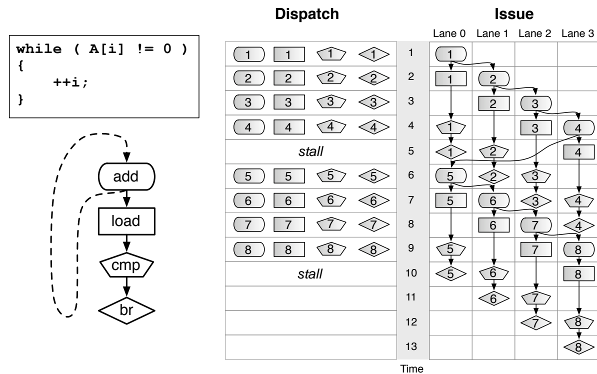
Figure 5: Zooming in on the dependence-based machine. The graph shows a cumulative distribution of the cycle-by-cycle reorder buffer occupancy ('rob') and aggregate issue queue occupancy ('lanes'). The 'heads' plot shows the occupancy of the 8 slots at the front of the lanes (*i.e.* one slot from each lane). All data is averaged across the 12 SPEC Integer benchmarks.

instruction in them at any one time. Recalling the steering heuristic described in Section 2.1, this points to a scenario in which dataflow dependences are causing instructions to be spread "shallow and wide" across the lanes rather than to be stacked up behind one another within the lanes. For the same reason, it will frequently be the case that the next instruction to be dispatched will not depend on any of the instructions at the tail of the lanes, requiring that steering logic stall until a free lane becomes available.

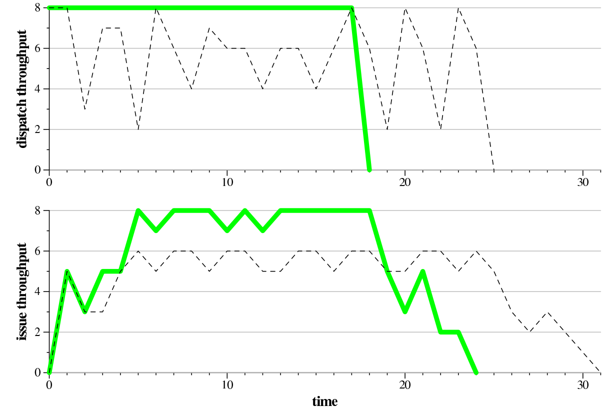
3.2 Accounting for dispatch stalls

The steering behavior described above might be caused by a number of factors. For example, the shape and distribution of ILP in the instruction stream are clearly both important. But other factors such as the dynamic distance between producers and their consumers, branch mispredictions and cache misses all have a part to play. Nevertheless, it is properties of dataflow that have the first-order effect. Simply put, the problem is that dataflow does not decompose into just a few dependence chains. Rather, it tends to

[1] This data is consistent with observations made by Michaud and Seznec [5], who found that a dependence-based window with n FIFO buffers is equivalent (from an IPC performance point of view) to a monolithic window with $2n$ entries. At 25% occupancy, our 8-laned window is averaging about 16 instructions in-flight at any one time.



(a) Hypothetical code



(b) Sample from the `twolf` benchmark

Figure 6: Stalls throttle loop initiation. The figure on the left shows how a hypothetical loop would execute on a dependence-based machine. The loop's source code and corresponding static dataflow appear to the left of a timing diagram showing dispatch and issue of 8 successive iterations (instructions are annotated with the iteration to which they belong). The timing diagram models the operation of a machine with 4-wide fetch and execute bandwidth and 4 lanes in the execution core; loads all hit in the cache and have a 2 cycle load-to-use latency. Dispatch stalls arise every 5 cycles because lanes are consumed by dispatch faster than they are cleared by issue. The two graphs on the right show this effect at work in real code. Each shows cycle-by-cycle dispatch (top) and issue (bottom) throughput for a short trace from 9 successive iterations of the inner loop in function `dbox_pos_2` in the `twolf` benchmark. The thicker, pale line tracks dispatch and issue activity on a machine with a monolithic window; the dashed line tracks that on a dependence-based machine.

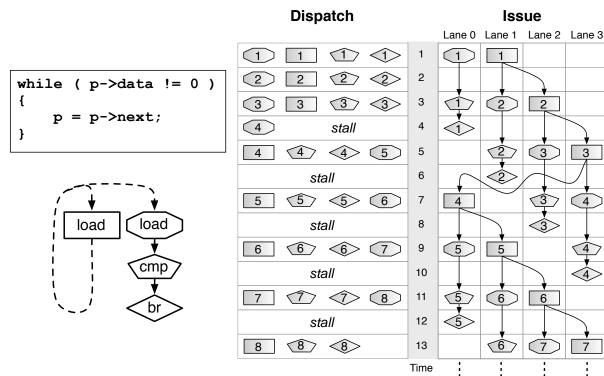
comprise a large number of chains, most of them very short. We now show that this can be accounted for by the manner in which small loops dynamically unwind in the window. In the next two sub-sections, we describe this process in detail, together with its principal repercussions for performance.

3.2.1 Converting execute- into fetch-criticality

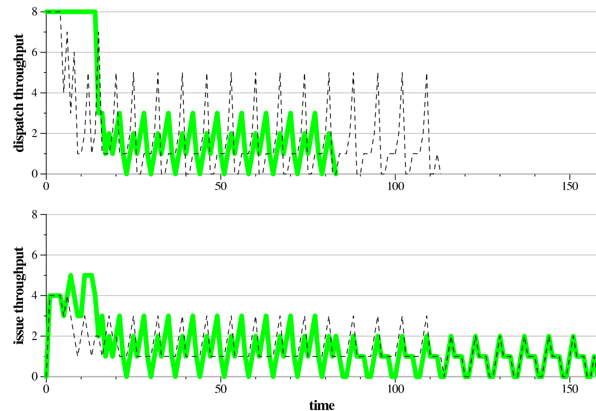
Figure 6(a) illustrates a typical "spine-and-ribs" dataflow structure induced by the dynamic unrolling of a small loop. The spine consists of induction variable computations (the add instruction), and the ribs the intra-loop work that branches off the spine (the load-compare-branch sequence). Although it is somewhat contrived, the example distills a basic process at work in the dependence-based machine: as a loop is fetched into the window, it decomposes into small dependence chains (the ribs), causing successive iterations to spread *across* lanes. This behavior arises because dataflow *diverges* — and so spawns a new dependence chain — at each induction variable update. And because loop iterations are fetched faster than they can be executed, this rapid consumption of lanes incurs one dispatch stall every 5 cycles. Those stalls translate into a sustained IPC of 3.2 for the hypothetical code sequence — a 20% loss relative to the peak IPC of 4 that a monolithic machine would achieve. This problem arises simply because the dependence-based machine cannot deliver instructions into the window fast enough to expose the available ILP. *It converts execute-criticality into fetch-criticality*.

Figure 6(b) shows this phenomenon at work in real code. It contrasts dispatch and issue activity over successive iterations of a hot loop in the `twolf` benchmark. That loop has a very high initiation rate and enough ILP to keep the machine operating at close to peak performance. The top graph shows that instruction supply rate in the dependence-based machine (dashed line) is being throttled relative to that in the monolithic machine (solid line), with the latter delivering the trace into the window much quicker than the former. This difference in instruction throughput prevents the available ILP from being exposed to the dependence-based machine's issue logic, causing it to take longer than the monolithic machine to execute the trace (bottom graph).

Although this effect is certainly the principal cause for most of the IPC losses we observe, it is not, in general, a sufficient condition for them. As a counterpoint to previous examples, the hypothetical loop in Figure 7(a) shows that peak performance can be achieved even in the presence of many stalls. In this case, the loop's induction variable updates have longer latency, so the peak initiation rate and, hence, the available ILP, is now lower. Thus, while lanes are consumed very rapidly by this loop, the ensuing stalls do not manifest themselves as performance loss; they are not exposed on the critical path. Figure 7(b) shows that real code exhibits exactly this kind of behavior. In this case, the trace is from a hot loop in the `gzip` benchmark, where available ILP is very low. Although dispatch stalls are clearly throttling instruction supply relative to the monolithic machine (top graph), the lack of available ILP means the throttling effect is benign: both the monolithic and the dependence-based machine finish executing the trace at exactly the same time (bottom graph). That this loop is so hot in `gzip` explains why the dependence-based machine performs very well on this benchmark (recall Figure 3) — in spite of a very large number of steering-induced stalls.



(a) Hypothetical code



(b) Sample from the `gzip` benchmark

Figure 7: Front-end stalls can be benign. Like Figure 6, the diagram on the left shows a hypothetical code example executing on a 4-wide dependence-based machine. In this case, the throttling effect occurs in the shadow of latencies imposed by low-ILP code. The graphs on the right show how this process affects dispatch (top) and issue (bottom) in a trace of 22 iterations of the main loop in the `updcrc` function in the `gzip` benchmark. Once again, the thicker, pale line tracks dispatch and issue activity on a machine with a monolithic window, and the thinner, dashed line tracks that on the dependence-based machine.

In general, the extent to which execute-criticality will be converted to fetch-criticality is determined by two factors. The first is the rate at which instructions can be delivered into the window, which is determined by how quickly lanes are consumed by each iteration of the loop and by how long each iteration takes to complete its execution. Loops with a large amount of ILP and with long-latency operations within each iteration will consume lanes quickly and will block each lane for longer. The second factor is the peak ILP that is available from dynamically unrolling the loop, and how many times the unrolling must occur in order to expose that ILP. When loop initiation rates are low, fewer iterations will have to be unrolled to expose all the available ILP. This mitigates the instruction supply problem.

This characterization implies that equipping the dependence-based scheduler with more lanes would alleviate its problems. Indeed, in experimenting with more lanes (while maintaining the same aggregate issue queue size), we found that performance steadily increases. But it is not until the machine is equipped with 24 lanes (each 3-deep) that performance losses drop below 5%. This is clearly not a reasonable design point. Steering 8 instructions to any one of 24 lanes each cycle is complex and, more importantly, exposing 24 instructions to the select logic, and requiring that all 24 be able to check operand availability each cycle, renders the design not altogether different from a monolithic scheduler.[2]

A more compelling approach would be to diminish front-end throughput relative to the number of lanes. For example, a 4-wide front-end dispatching to 8 lanes would reduce the rate at which lanes are consumed by steering relative to the rate at which they are made available by execution. This is precisely what Kim and Smith did in their ILDP work on accumulator-oriented machines [3, 4]. Our own evaluation of such a configuration yielded a penalty of about 15% — still rather severe, but a substantial improvement over the 30% losses suffered by an 8-wide front-end.

3.2.2 Exposing non-critical work as fetch-critical

Implicit in the above discussion was an assumption that the instruction traces under discussion were execute-critical. That is, the stalls imposed by lanes were deemed benign or deleterious depending on their impact on the machine's ability to extract ILP. However, machines don't always operate in an execute-critical regime, and the ILP extracted from a given instruction trace does not necessarily have any effect on runtime. Instruction supply rate is also important, and here, too, the impact of lanes can be severe.

The `twolf` example we discussed earlier is, in fact, a good example of a piece of code that is not execute-critical. It so happens that the small loop we traced in Figure 6(b) is nested within another loop that contains critical dataflow on the backward slices of cache misses and branch mispredictions. Since the inner loop, though executed often, has a low and predictable trip count, it constitutes a sequence of instructions that must simply be fetched into the window in order to reach the more important instructions that follow it; the inner loop's execution can safely continue in the shadow of the surrounding, long-latency operations. Because the dependence-based machine throttles the rate at which that fetching can occur, the inner loop becomes a barrier to forward progress. An analogous problem is encountered by `gcc`, which suffers very severe performance losses

[2] Each increment in lane count renders the design increasingly similar to the monolithic baseline machine, whose 64-entry window can be viewed, abstractly, as a dependence-based one with 64 lanes, each 1-deep.

on the dependence-based machine (recall, again, Figure 3). This benchmark spends a significant fraction of its time in a `memcpy` loop, which has been unrolled by the compiler so that each fetch cycle delivers a large amount of ILP into the window. Lanes are therefore consumed very rapidly, and dispatch stalls are commensurately frequent. Because this loop suffers so many cache misses, the throttling of instruction supply in this case translates into a serialization of cache misses and a commensurate reduction in the amount of exposed MLP.

We stress that this barrier effect can apply regardless of the IPC achieved within the blocking region. For example, had the `gzip` loop in the earlier example also had a low trip count, and had it also been nested in an execute-critical region, the dependence-based machine's ability to match the IPC of the monolithic machine would not be of much benefit. It is the throttling of instruction supply — shown by the top graph in Figure 7(b) — that matters in this case.

4 Accounting for discrepancies

Now that we understand the main processes at work in a dependence-based machine, we can explain why the two studies produced such disparate results. As we mentioned earlier, in both our own study and that conducted by Palacharla *et al.*, an attempt is made to evaluate the dependence-based scheduler by incorporating it into a machine that stresses its execution core to the fullest. This objective is reflected in each study's choice of baseline machine parameters. Comparing the tables in Figures 2 and 3, it is clear that both simulators model fairly aggressive instruction and data supplies, and both provide ample execute bandwidth. However, several differences stand out.

- *Branch prediction.* Whereas we use an aggressive tournament branch predictor, the original study modeled only a 12-bit *gshare* predictor.
- *Memory disambiguation.* We modeled ideal memory disambiguation with an unbounded load/store queue, while the original study was more conservative, requiring that loads wait for all prior store addresses to resolve.
- *Latencies.* The original study assumed all instructions execute with unit latency, whereas we used different latencies for different instruction types; L1 cache miss penalties were also different.

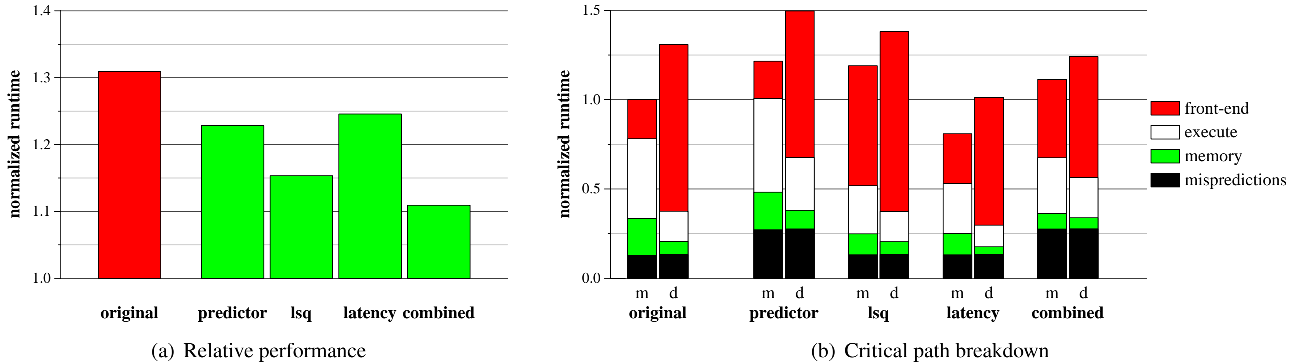
We can show that the above differences are, together, sufficient to account for the discrepancy in results between the two studies.

We do so by working backwards from our own machine configuration, successively changing the microarchitecture until we reach a configuration similar to the original study.[3] Figure 8(a) shows how relative performance is thereby affected. All microarchitectural changes reduce the penalty incurred by a dependence-based scheduler, so all have a positive effect on the scheduler's apparent efficacy. Among them, the memory disambiguation constraint is clearly the most important, since it alone cuts the relative performance loss in half (from 30% to 15%). Taken together, the various changes reduce a 30% loss to about 11%, bringing our results close in line with those of the original study.

To understand why relative performance changes in this way, we can once again make use of critical path analysis. Figure 8(b) plots the critical path breakdown for the monolithic and dependence-based machines when each of the changes is applied to the underlying machine model. The most obvious trend is an overall increase in the extent to which fetch-criticality contributes to the *monolithic* machine's critical path. This effect is most pronounced with the introduction of the more conservative memory disambiguation policy (the 'lsq' bar), which induces front-end stalls when the LSQ fills, or when loads and their forward dataflow slices back up in the issue queue (because loads must now wait for all older store addresses to resolve). The dependence-based machine is in this case hardly affected because those new front-end constraints are introduced into a pre-existing fetch-critical regime: their effects are *shadowed*. Modeling shorter instruction latencies (the 'latency' bars) naturally improves performance in both machines, but in the monolithic machine this benefit is countered by a shift to fetch-criticality, a result of the now faster execution core exposing an underlying instruction supply constraint. In contrast, the shorter latencies mean the dependence-based machine manages to clear out its lanes faster, so its fetch-criticality is in fact reduced. The change in branch misprediction strategy (the 'predictor' bars) has a very different effect. Naturally, both machines see an increase in time spent recovering from mispredictions. However, in the monolithic machine, there is also an increase in the time spent executing instructions because the additional mispredictions have a serializing effect, hiding ILP that was previously being exploited. The dependence-based machine is less susceptible to this because it was simply not exploiting that ILP in the first place.

Given that the original study was conducted about a decade ago, it might be argued that our analysis merely exposes a problem in mapping the dependence-based design into a more modern context. However, this is not the case. The original study deliberately attempted to model an underlying microarchitecture in which the execution core

[3] We were unable to discover, from the literature [6, 7] and from the authors, the size of the load/store queue (LSQ) used to impose the memory disambiguation policy. We chose 32 as our best estimate of its size. Palacharla *et al.* used SimpleScalar Version 1.0 [1] in their empirical work, scaling the default RUU size by a factor of 4 to model a 64-entry issue window; modeling a 32-entry LSQ scales its default size by the same factor.



(a) Relative performance

(b) Critical path breakdown

Figure 8: The impact of machine parameters. The graph on the left shows how performance of the dependence-based machine relative to that of the monolithic baseline is affected by successive changes to the underlying microarchitecture (of both machines). The bar labeled 'original' represents the configuration we started with in Section 2 (see Figure 3). The 'predictor' bar replaces our tournament branch predictor with the same gshare predictor used in the original study. The 'lsq' bar imposes the original study's memory disambiguation constraint, which prevents loads from issuing until all older store addresses are resolved. The 'latency' bar shows the effect of modeling unit-latency instructions and a smaller L1 miss penalty. The net effect of all these factors is shown by the 'combined' bar. The graph on the right shows the effect of these changes on the critical path breakdown for both the monolithic baseline ('m') and dependence-based ('d') machines. In this case, all bars are normalized to the leftmost one. Data is in all cases averaged across all the 12 SPEC Integer benchmarks.

was stressed to the fullest. Our analysis shows that this was probably not happening. Rather, these results imply that the good performance originally reported is most likely due to the dependence-based scheduler being incorporated into a machine that was barely exercising its execution core. One way to confirm this is to severely constrain the *monolithic* machine's execution core by reducing its issue queue size from 64 to 16 entries. Doing so for the constrained monolithic machine (from the 'combined' bars in Figure 8) results in a very small performance loss of only 8%, confirming that the execution core is not the performance bottleneck. In contrast, likewise reducing issue queue size in our own monolithic machine (from the 'original' bars in Figure 8) results in a 25% performance loss.

5 Conclusion

The dependence-based scheduler is appealing in a number of respects. Not only does it offer implementation benefits, but embedded in its good performance figures is the compelling result that basic properties of dataflow can be exploited to build a very simple scheduler. However, our evaluation of the new scheduler paints a very different picture. We find that the machine suffers from an serious instruction supply problem, one arising because of dispatch stalls imposed by the dependence-based steering policy. Those stalls can, in turn, be ascribed to basic properties of dynamic dataflow, which cause instructions to spread "shallow and wide" across the machine's lanes rather than stacking up within them. The net result is that the machine's effective issue queue size is determined by the number of lanes, not by the depth of each one. For example, 8 lanes provide the machine with an effective capacity of little more than 16 instructions. Thus is lost one of the most important benefits of a conventional design: the ability to *buffer* instructions in order to *decouple* dispatch from the order and rate at which instructions are executed. In this respect, a monolithic window is ideal because it facilitates dispatch as long as *any* one of its entries is available; it imposes only its finite capacity as a constraint on instruction supply. A dependence-based machine, in contrast, renders the window's ability to buffer work subject to the shape of dynamic dataflow.

Diagnosing these problems also permits us to explain why the dependence-based machine performed so well in its original evaluation. It appears there was a fortuitous interaction between constraints already present in the baseline machine and those introduced by the new scheduler. These caused the scheduler's instruction supply problems to be hidden behind pre-existing bottlenecks. In short, the original study was not exercising the new scheduler, so its performance problems were not exposed.

An obvious question now arises as to potential of alternative steering policies that do not stall when a suitable slot cannot be found. However, it should be borne in mind that the policy proposed by Palacharla *et al.*, which ensures that only dependent instructions are slotted behind one another in each lane, is important for rendering the in-order issue constraint benign. Any alternative that does not stall will perforce expose instructions to in-order execution. A proper exploration of these issues is unfortunately beyond the scope of this work. That said, our initial exploration of a few alternative policies leads us to conclude — as others have done [5] — that the original heuristic proposed by Palacharla *et al.* is, in fact, a good one.

Our ability to reach all of the foregoing conclusions is underpinned by *critical path analysis*. Fields's critical path

model [2] has proved indispensable in this work, both as a tool for characterizing performance (by means of critical path breakdowns) and as a basic conceptual model in which to reason about machine behavior (notions of fetch- and execute-criticality, for example). This is particularly true in terms of empirical evaluation of a machine's performance. Researchers tend to have a feel for what constitutes a reasonable IPC figure, but vagaries in experimental setups can cloud this already vague and informal means for reasoning about empirical work. A critical path breakdown can *complement* an IPC figure by serving as a "signature" of a machine's behavior. This is needed to demonstrate, first, that a given machine does not spend an inordinate amount of time in any one part of the pipeline and, second, to confirm that performance does indeed derive from the factors presumed to be responsible. The original work on the dependence-based machine provides a good example of how IPC, alone, is too coarse a metric for understanding the behavior of a machine. Had a tool for critical path analysis been available in the original study, the dependence-based machine's instruction supply problems would have been immediately exposed.

Acknowledgements

This research was supported in part by NSF CCF-0429561, NSF CAREER award CCR-03047260, and a gift from the Intel corporation.

References

[1] D. Burger, T. Austin, and S. Bennet. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, University of Wisconsin-Madison Computer Sciences Department, 1996.

[2] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[3] H.-S. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–82, May 2002.

[4] H.-S. Kim and J. Smith. Dynamic binary translation for accumulator oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 25–35, March 2003.

[5] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 27–36, January 2001.

[6] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin–Madison, 1998.

[7] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[8] J. Stark, M. Brown, and Y. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 57–66, December 2000.