Fundamental Performance Constraints in Horizontal Fusion of In-order Cores

Pierre Salverda Craig Zilles

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

Abstract

A conceptually appealing approach to supporting a broad range of workloads is a system comprising many small cores that can be fused, on demand, into larger cores. We demonstrate that using in-order cores for this purpose, even under idealized assumptions about fusion-related overheads, would introduce fundamental obstacles to achieving good performance — obstacles that are not present when out-of-order cores are used. Matching the performance of modern dynamically-scheduled designs demands that a fused machine be able to simultaneously manage a large number of active dataflow chains, many more than the amount of ILP typically extracted from the code. When it is in-order cores that are fused, this requirement, in turn, demands either that the active dataflow chains be carefully interleaved among the available issue queues, or that enough cores be provided for them to reside at distinct queues. Using an abstract model for reasoning about the performance of these machines, we show that the former option is fundamentally hard, in the sense that it necessitates instruction steering hardware that would be too complex to build. The latter option would demand so many cores that the machine would be overwhelmed by fusion-related overheads. In short, if the goal is to match the performance of modern dynamically-scheduled machines, fusion of in-order cores is not a very compelling approach; either a fundamentally new method for fusing cores is needed, or some form of out-of-order capability must be provided at the constituent cores.

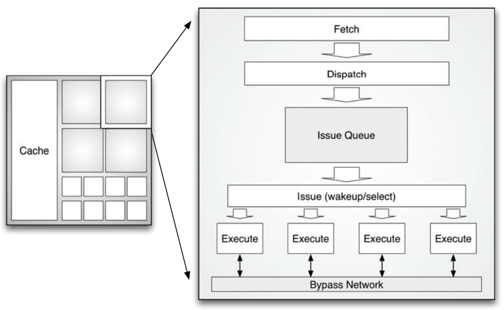
1 Introduction

It appears two classes of programs will dominate in future client system workloads. The first comprises the massively-parallel, computationally-intensive programs in areas such as graphics, physics, signal processing, and recognition, mining and synthesis (RMS). The second constitutes the very large body of remaining applications that resist parallelization. Programs in the first category may well contribute most of the instructions committed by the processor, but they will likely constitute only a small fraction of the total programs that must be supported: the bulk of applications fall into the second category. Each class of program places very different demands on the processor. The first benefits from a wealth of narrow, perhaps multithreaded, cores of modest clock speed, with wide SIMD functional units and a high-bandwidth memory system. The second prefers a single high-frequency, wide-issue, aggressively-speculative, out-of-order processor with a low-latency memory system — requirements that are clearly at odds with those of the first category. Designing a single chip to handle both application groups therefore constitutes a challenging problem.

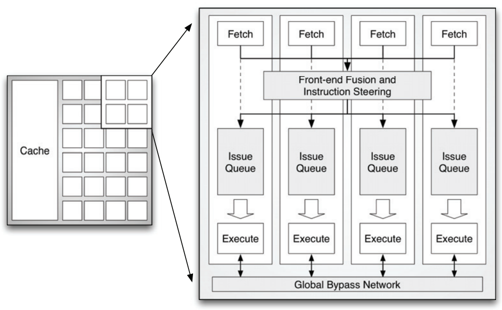
A number of solutions have been proposed in the literature. One well-studied technique is the heterogeneous chip multiprocessor (CMP) [11, 12], an example of which is depicted in Figure 1(a). This machine comprises a small number of aggressive out-of-order superscalar cores, plus a sea of small, efficient, throughput-oriented cores. Though each type of core is designed specifically for the class of workload it is intended to execute, this approach has two drawbacks. First, a chip vendor must now design two (or more) cores for a single product release. Second, any applications that fall between the two extremes on the workload spectrum may be poorly served.

More recently, an alternative approach has been proposed in which a homogeneous many-core CMP is extended to support on-demand *aggregation*, or *fusion*, of small processor cores into a large out-of-order uniprocessor [6, 8]. Figure 1(b) depicts this conceptually appealing idea. Its principal benefit is that it requires the design of just one simple core, yet is flexible enough to cater to a wide range of thread- and instruction-level parallelism (TLP and ILP, respectively) in the workload.

A notable factor in recent fusion work is the use of small dynamically-scheduled processors as the basic unit of composition. However, the many-core designs being developed and announced by industry, such as Sun's Niagara [16], Intel's recently announced Larrabee, ClearSpeed's CSX600 [4], the Cell's SPEs [7], Ageia's PhysX [1] and GPUs, are all in-order substrates. This is a reflection of the target domain for those machines, where massively-parallel, compute-intensive workloads are the norm. In those programs, out-of-order execution is not required, neither for scheduling (since narrow machines are used), nor for latency tolerance (since multi-



(a) Heterogeneous CMP



(b) Fusion of simple cores

Figure 1. **Two approaches to dealing with workload diversity.** On the left, a heterogeneous CMP includes a few powerful cores for sequential workloads and many simple cores for massively-parallel workloads. The homogeneous CMP on the right comprises many simple cores, subsets of which can be fused (4 in this case) into a distributed, wide-issue processor to support sequential workloads.

threading can be used instead). In fact, supporting out-of-order execution would only incur area and power overheads, reducing the chip's peak throughput potential.

The ideal design, therefore, would be a many-core substrate comprising in-order cores, but which supports fusion of those cores to synthesize out-of-order execution capabilities when they are needed. In such a framework, fused cores would support a restricted form of out-of-order execution, one in which the individual cores continue to execute instructions in-order, but which can slip relative to one another to achieve out-of-order execution overall. The question we tackle in this paper is whether this *slip-oriented out-of-order execution model* renders the machine versatile enough to deliver good performance on single-thread workloads. Appealing though it is, we find that its performance potential is fundamentally limited by basic properties of program dataflow. We will show that achieving good performance demands either very sophisticated schemes for distributing (steering) instructions among cores, the complexity of which would probably exceed that of conventional dynamic schedulers; or it demands the fusion of so many cores that fusion-related overheads would render the design impractical. In short, we find that fusion of small cores is not appealing if those cores support in-order execution only; some form of out-of-order capability is needed for effective synthesis of out-of-order execution in the aggregate.

We start our evaluation in Section 2, where we describe precisely what we mean by horizontal fusion of in-order cores, and by the slip-oriented execution model that arises in such a context. It is this restricted execution model that ultimately determines the viability of this class of machine, and so it is to its performance potential that we confine our attention. We therefore proceed to optimize away all the overheads that inevitably arise when otherwise independent cores are fused, thereby leaving just the per-core in-order issue constraint as the primary factor affecting performance. In Section 3, we then informally show that it is basic properties of dynamic dataflow that pose problems for slip-oriented machines. In so doing, we also provide the intuition behind two principal avenues for overcoming those challenges.

The first of those is *instruction steering*, the policy used to distribute instructions among the cores. Section 4 is devoted to this subject. We show that potential for better steering does indeed exist, but that implementation complexity imposes fundamental obstacles to achieving that potential. In fact, we show that *any* policy aiming to improve performance will *necessarily* be too complex to build. The key contribution of this paper, which underpins that result, is a conceptual framework for reasoning about performance of slip-oriented execution. We use that framework to show that achieving performance within 40% of a comparably-resourced dynamically-scheduled machine requires that each steering decision takes into account its potential to delay execution, both of the instruction being steered and of those yet to be steered. In effect, steering decisions must be governed by exact knowledge of when instructions will eventually execute. It is this requirement that is both necessary for good performance and fundamentally hard to achieve in practice. And it is this requirement that is ultimately imposed by the in-order issue constraint: if steering were instead targeting out-of-order cores, each decision would have fewer ramifications in terms of when subsequently steered instructions can execute.

The second strategy for improving performance, which we explore in Section 5, is adding more cores to the fused design. We find that increasing the number of cores relative to instruction fetch width steadily improves the ability to exploit ILP, even when a simple instruction steering policy is used. However, obtaining acceptable performance demands an inordinately large number of cores — 12–16 to match the performance of a conventional 4-wide out-of-order machine, for example. Aggregating such a large number of cores exacerbates fusion-specific overheads, not least of which is the cost of inter-core communication, which we show can quickly outweigh the benefits of more cores. We also show, however, that it is principally the addition of extra issue queues, not whole cores, that facilitates improved performance, an observation that leads us to consider designs with more than one in-order issue queue per core. While these can indeed match the performance of designs that fuse many more cores, they rely on a (comparatively modest) form of out-of-order execution at each core to sustain slip among the local in-order issue queues.

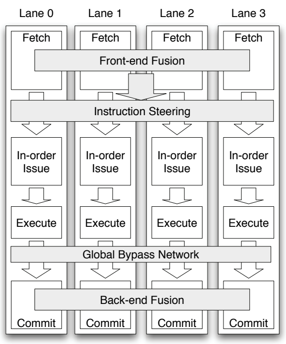
We want to emphasize that this paper does not merely show where we were or were not able to achieve compelling performance with horizontally fused in-order cores. Rather, our analysis explores *fundamental* aspects of performance in these machines, independent of artifacts of specific design choices. Our principal contribution is to delineate the design space, showing where solutions for good performance might be found, and, more importantly, where they simply cannot be found. We feel that a study of these fundamental issues is a necessary first step in this new area. And though our conclusions are, of course, subject to our assumptions about the underlying execution model, those assumptions are general enough to cover a wide range of designs. Short of fundamental changes to the way we fuse the in-order cores, our principal conclusion, which is that some form of out-of-order capabilities are necessary for synthesizing out-of-order execution overall, holds true.

2 Background

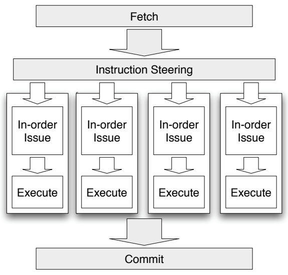
In this section, we describe the class of machine to which this paper is devoted. Section 2.1 explains what we mean by *horizontal fusion*, and by *slip-oriented out-of-order execution*, which is the particular execution model that arises when it is *in-order* cores that are horizontally fused. As we noted earlier, our focus is on basic principles, not on specific designs, so we idealize most of the overheads introduced by fusion, to the point that only the underlying slip-oriented execution model differentiates the fused machine from a conventional out-of-order superscalar. After briefly describing our experimental infrastructure in Section 2.2, we present initial performance figures in Section 2.3 to show that this restricted execution model is prone to some very severe performance problems.

2.1 Horizontal fusion of in-order cores

Figure 2(a) provides a high-level view of a machine that horizontally fuses four small in-order cores. We refer to each of the in-order cores within the aggregate as a *lane*, and to the aggregate itself as a *laned machine*. When fused, the individual lanes operate together as a single processor. In the front-end, each lane contributes to the fused machine's fetch bandwidth by supplying a portion of the fetch packet each cycle. Coordinating these otherwise disjoint instruction supply units requires the addition of a centralized structure, shown in the diagram as the front-end fusion logic. This potentially complex piece of machinery controls instruction fetch from distinct instruction caches, manages control flow prediction in some way, and orchestrates fetch redirection when branches are predicted (detected) to be taken (mispredicted) at one of the lanes. Once each fetch packet has been assembled, the front-end logic renames instructions and then *steers* (dispatches) them to an issue queue at one of the lanes, where they will eventually execute.



(a) Fusing in-order cores



(b) Idealized machine

Figure 2. **Laned machines.** The diagram on the left shows a high-level view of four in-order cores (lanes) being fused into a larger, 4-wide laned machine. New hardware required to coordinate the fused cores is shaded. The diagram on the right shows the idealized view we adopt for these machines, one in which all overheads introduced by the fusion-specific hardware are ignored.

Because lanes operate in-order, an instruction becomes eligible for issue when it reaches the head of its issue queue and when all its operands are available. Some of those operands will perhaps have been computed remotely at another lane, so some form of global communication mechanism is needed to distribute data among the lanes. Figure 2(a) shows a *global bypass network* for this purpose.

Instructions that have completed execution have their results committed, in program order, by centralized back-end fusion logic. This is needed because execution across lanes does not occur in lockstep, so instructions can complete their execution out-of-order. As a result, the set of instructions in-flight between dispatch and retirement constitute an instruction window, no different, in principle, from that maintained by conventional dynamically-scheduled designs. But the execution model supported within that window is a restricted one, with out-of-order execution arising solely through slip among the in-order lanes; execution is locally in-order, globally out-of-order. It is the IPC potential of this *slip-oriented out-of-order execution model* that forms the central focus of this paper.

The additional hardware required to coordinate otherwise independent cores inevitably introduces a number of overheads. For example, fusion logic necessarily lengthens the front-end pipeline, thereby increasing the branch misprediction penalty, and inter-lane communication introduces a global communication penalty, which effectively lengthens any dataflow chains that cross lane boundaries. *We will largely ignore these overheads in this paper.* We do so deliberately. We want to explore performance constraints arising solely as a result of slip-oriented execution, and which are therefore inherent to the laned machines; modeling design-specific overheads would merely cloud that analysis. We therefore make the following very optimistic assumptions about the implementation. The resulting idealized machine, which closely resembles a clustered superscalar [2, 3], is depicted in Figure 2(b).[1]

1. **Front-end**. To capture just the effects of register rename, we assume a laned machine's front-end is the same as that of an equal-width dynamically-scheduled machine. Coordination of control flow across lanes and instruction steering logic introduce no additional overheads.
2. **Execution core**. Global communication is free: transferring register values among lanes occurs with zero latency and unbounded bandwidth. We assume memory disambiguation occurs in a centralized manner and without any additional delays. We also place no bounds on issue queue capacity at each lane, so only the ROB places a limit on the number of in-flight instructions.
3. **Memory hierarchy**. We assume a single data cache for all cores comprising a laned machine, and that the cache has sufficient bandwidth to support as many lanes as are involved in the fusion. If each in-order core originally has its own L1 data cache, we assume no overheads are incurred in aggregating those into a single, shared cache.

We will, in some cases, show the impact that a global communication penalty would have on performance, but only so that we can quantify the extent to which (a fraction of) our idealized assumptions are benefiting performance.

In spite of these idealized assumptions, the laned machines are prone to performance problems, the severity of which we now demonstrate with an initial set of experiments. First, however, we briefly describe the experimental infrastructure we make use of in our work.

2.2 Experimental framework

Throughout this paper, we evaluate two types of laned machine, one with a 4-wide front-end and one with an 8-wide front-end. For each of those, we explore configurations in which the in-order cores support 1- and 2-wide issue. For convenience, we refer to the resulting configurations using the nomenclature Fw–($L \times I$w), where F denotes the machine's effective front-end width, L the number of lanes it has, and I the issue width of each of those lanes. We will use the shorter $L \times I$w notation when front-end width is clear from context. We will not, in general, require that $F = L \cdot I$. More specifically, we will permit configurations in which the number of lanes exceeds the front-end width we actually use. For example, the 4w–(8×1w) machine fetches 4 instructions per cycle, but steers them among 8 1-wide cores; only 4 of the 8 instruction supply units are being used in this configuration.

[1]General though this view is, it does not cover a machine like Voltron [18], which relies fundamentally on the compiler to orchestrate when and how fusion occurs, as well as to statically map instructions to lanes. We confine our analysis to schemes that are invisible to the compiler.

	2-wide	**4-wide**	**8-wide**
Fetch	perfect instruction cache; perfect uncond. predictor; tournament cond. predictor.	same.	same.
Front-end	2-wide, 5 stages to dispatch	4-wide, 8 to dispatch.	8-wide, 12 to dispatch.
Window	64-entry reorder buffer; 32-entry unified issue queue.	128 ROB; 32 IQ.	256 ROB; 64 IQ.
Execute	2 int, 2 fp; 1 memory port. latencies similar to 21264; ideal mem. disambiguation.	4i / 4f / 2m. same.	8i / 8f / 4m. same.
Memory	L1: 64KB, 4-way, 2-cycles; L2: 8MB, 8-way, 12-cycles; DRAM: 300 cycles.	same.	same.
Back-end	2-wide.	4-wide.	8-wide.

Table 1. **Monolithic baseline machine configurations.**

To gauge the efficacy of the laned machines, we compare their IPC to that of 2-, 4- and 8-wide out-of-order superscalar designs, aiming thereby to understand where in the spectrum of out-of-order performance capabilities the laned machines reside. Throughout this work, we will refer to those baselines as the *monolithic* machines. Table 1 enumerates their main architectural parameters.

Since we are interested specifically in laned machines in terms of their potential to perform well on sequential workloads, we confine our analysis to the SPEC2000 Integer benchmark suite. All programs were compiled for the Alpha ISA using the DEC C Alpha compiler (V5.9-005), with peak optimization enabled, but with no profile feedback. All of the data we present is obtained from cycle-accurate simulation of three 100-million instruction traces, one each at 3-, 5- and 8-billion instructions into the benchmark's run. Due to space considerations, we will present average performance results only, but assure the reader that all benchmarks observe the same trends as the average.

2.3 Performance evaluation

As a first step toward understanding the potential of laned machines, we compared the performance of a few basic configurations against that of our monolithic machines. To manage instruction steering in the laned machines, we used the *dependence-based steering policy*, which was developed by Palacharla *et al.* for distributing instructions among the FIFO buffers in their complexity-effective scheduler [14]. That same policy, slightly modified, was adopted by Kim and Smith in their ILDP work, where it was used to steer instructions in a clustered microarchitecture comprising 1-wide in-order execution units [9, 10]. Though recent work [15] has exposed some problems with this steering policy (a subject to which we will return in Section 4), it remains the best published scheme for distributing instructions among in-order execution units, and therefore a good starting point for our study.

We implemented the dependence-based steering policy for four basic laned machine configurations: the 4w–(4×1w) and the 8w–(8×1w) machines, which fuse 1-wide in-order

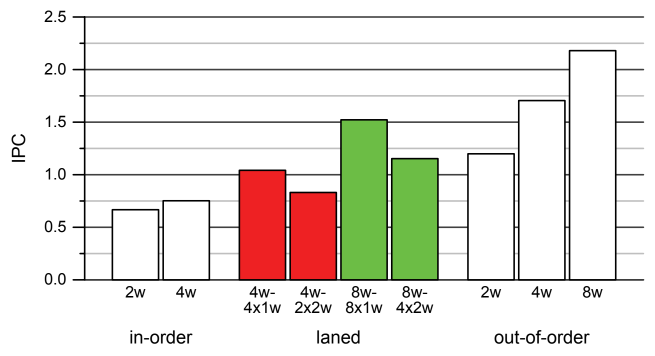


Figure 3. **Laned machine performance.** Each bar shows harmonic mean IPC across the SPEC2000 Integer benchmarks for a variety of machine configurations: 2 in-order superscalars, 4 laned machines, and 3 out-of-order superscalars. Table 1 enumerates the configuration parameters for the out-of-order machines. The in-order machines are configured similarly in terms of their memory hierarchy and issue capabilities, but their front-end pipelines are shorter.

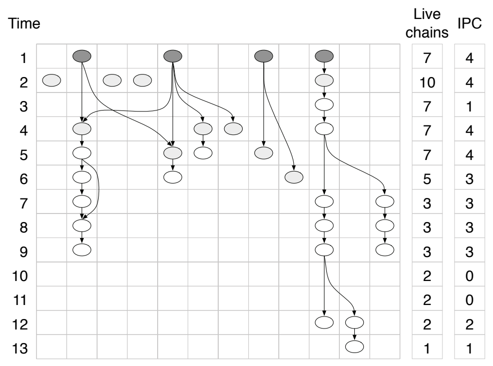


Figure 4. **Dataflow comprises many short chains.** The timing diagram shows the schedule effected by our 4-wide monolithic machine on a fragment of the `gap` benchmark. All instructions were resident in the machine's issue queue at the start of the timing, and left it at the indicated issue times. The lightly shaded instructions constitute the set of *live chains* active in the window after the issue, in cycle 1, of the dark instructions.

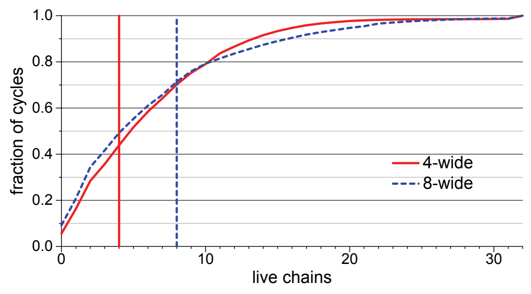


Figure 5. **Live chains.** The graph shows the cumulative distribution of cycle-by-cycle counts of live-chains active in the issue queues of our 4- and 8-wide monolithic baselines. The vertical lines highlight the fraction of the distribution that is covered by 4 and 8 lanes. Data is averaged across all 12 SPEC Integer benchmarks.

cores; and the 4w–(2×2w) and the 8w–(4×2w) configurations, which do likewise with 2-wide in-order cores. Figure 3 compares their performance to that of various monolithic designs. The laned machines improve on the performance of superscalar in-order designs, but they fail to match the out-of-order machines. Both the 4-wide and 8-wide laned machines fall well short of an equal-width monolithic out-of-order design, being about 1.6 and 1.4 times slower, respectively. In fact, the laned machines perform more like a much smaller out-of-order machine. Both of the 4-wide laned machine configurations fail to match even the 2-wide out-of-order machine; only one of the 8-wide laned machines manages to do so, but both fall short of the 4-wide out-of-order machine.

3 Understanding the challenges

Since we do not model any overheads, the above performance results are ultimately attributable to slip-oriented execution. In this short section, we informally describe the key problems this restricted execution model introduces and, in so doing, provide the intuition behind the two avenues for their mitigation that we will be exploring in the remainder of the paper.

3.1 Properties of dataflow

The slip-oriented execution model constitutes a fundamental departure from the type of out-of-order execution provided by conventional dynamically-scheduled machines. The Alpha 21264, for example, approximates, within the scope of its instruction window, a *dataflow-oriented execution model*, where only dataflow dependences constrain instruction execution. By contrast, slip-oriented execution introduces, in addition to dataflow constraints, an in-order issue constraint, which it imposes on subsets of all in-flight instructions.

This additional constraint interacts poorly with basic properties of dynamic dataflow. In general, dataflow is not uniform in shape: not all instructions execute with unit latency, and ILP is distributed unevenly in the instruction stream. As a result, sustaining an IPC of N generally requires simultaneously managing more than N chains of dataflow in the window. Figure 4 shows this effect at work in a fragment of the `gap` benchmark. Immediately after cycle 1, there are 10 *live dataflow chains* — 10 instructions which have no dataflow predecessors in the window. Not all of those are ready to issue at this time, so the achievable IPC is well below 10; but they will soon become ready. In a dataflow-oriented execution model, each live instruction will be able to issue as soon as it becomes ready, since it is only dataflow that constrains its execution. The same effect can be achieved in a slip-oriented model only if each live instruction reaches the head of an issue queue by the time it is data ready. That, in turn, demands either having enough lanes to buffer all the live chains, or it demands carefully interleaving the live chains among the available lanes. The former is an expensive prospect given the data in Figure 5. A 4-wide machine, for example, would need 14 or more lanes if it is to have sufficient buffering more than

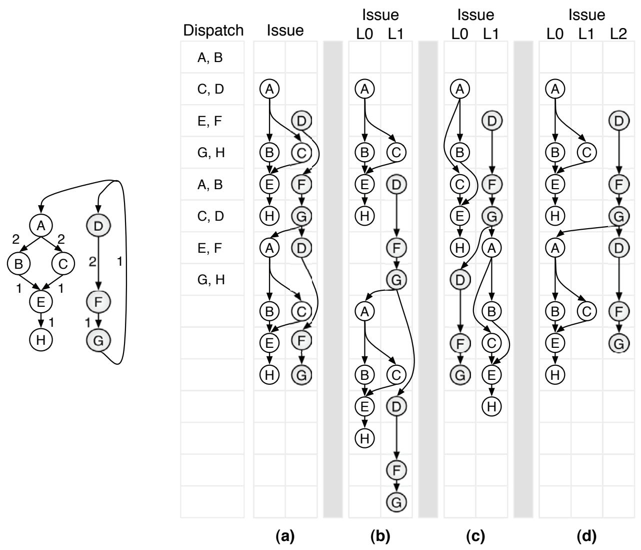


Figure 6. **The impact of lanes.** The static dataflow diagram on the left shows the body of a small, hypothetical loop. Dataflow edges are labeled with the latency of the producing instruction. The timing diagrams — (a) through (d) — show dispatch and issue of two successive iterations of that loop on various machines, all of which have 2-wide front-end and issue bandwidth. Diagram (a) shows the schedule obtained by a conventional out-of-order machine. Diagram (b) shows the operation of a 2×1w laned machine when instructions are steered using a simple dependence-based policy. Immediately to its right, diagram (c) shows the schedule obtained by a more sophisticated steering policy that interleaves live dataflow chains. Diagram (d) shows the operation of the same dependence-based steering policy used in (b), this time with an additional lane made available to it.

90% of the time. The latter is a patently hard problem, especially since the right interleaving must be effected at instruction dispatch time, well before execution times are known. In the next subsection, we make these ideas concrete by way of an illustrative example.

3.2 An illustrative example

Figure 6 uses a hypothetical code example to distill the key factors that underlie the performance problems encountered by laned machines on real code. The diagram shows the execution of a small loop on machines with 2-wide fetch and execute bandwidth. Assuming that no branch mispredictions occur, the peak performance achievable on that loop is 1.6 instructions per cycle. The timing diagram in Figure 6(a) shows that a conventional out-of-order machine can sustain that execution rate. The flexibility afforded by a dataflow execution model is key to its ability to do so: instructions are able to execute as soon as their operands are ready; the order in which they enter the window, and their location in the issue queue, has no bearing in this regard. By contrast, the laned machine shown in Figure 6(b) performs much worse because its in-order issue constraint serializes ILP in the window. Specifically, instruction `D` (and hence `F` and `G`) in lane `L1` is prevented from executing when its operand is ready because it is blocked while instruction `C` waits for its operand from `A`. The net effect is a complete serialization of successive loop iterations, reducing sustained ILP to just above 1.

A more judicious allocation of instructions to lanes can overcome these problems. It is the allocation of instruction `C` to lane `L1` in Figure 6(b) that is responsible for the performance losses, since it is this instruction that consistently blocks the `D`–`G` chain. The schedule shown in Figure 6(c) shows how steering instruction `C` to `L0` exposes the ILP available via the `D`–`G` chain. Although the `A`–`H` chain now takes slightly longer than necessary to execute, the resulting IPC matches that of the out-of-order machine. This example makes it clear that steering can have a profound impact on performance, but also that good steering decisions are not necessarily obvious. Indeed, the decision made in Figure 6(c) required knowing, in advance, that instruction `C` should be deliberately delayed in order to make room for instructions `D` through `G`. Even harder, it required knowing that the delay imposed on `C` in sending it to lane `L0` would be small enough not to outweigh the benefits of doing so. We will show in Section 4 that any steering policy that can effectively interleave live chains in this manner will be too complex to build.

An alternative means for improving performance is shown in Figure 6(d). In this case, the addition of a third lane to the machine permits even a simple steering policy to match the IPC of the monolithic machine. This is possible because the `D`–`G` dataflow chain can be steered in such a way that it is no longer exposed to the in-order issue constraint. It is not the added issue bandwidth that helps in this case, however, but rather the improved ability to expose the live chains to the issue logic: adding lanes increases the chances that an instruction will be at the head of a lane when it becomes data ready. Of course, adding more lanes to a machine is not a very cheap or efficient means for improving its ability to exploit ILP, especially when all the overheads of fusing cores are taken into account. We explore this issue in Section 5.

4 Instruction steering

In this section, we focus on fused designs with a modest number of lanes. Our objective is to determine if it is possible to develop a steering scheme capable of outperforming the dependence-based policy evaluated in Section 2.3. We tackle this problem with the benefit of hindsight, having already explored a large number of alternative heuristics, all of which failed to do any better. In the process, we reached the conclusion — as have others [13] — that this is not a problem for which a simple heuristic will suffice. In this section, we put that informal claim on a rigorous footing. In Section 4.1, we introduce an abstract model for reasoning about fundamental requirements for making good steering decisions. That model, which is agnostic to microarchitectural details and to steering policy specifics, permits us in Section 4.2 to distill the key features required of *any* policy that aims to effectively distribute instructions among in-order issue queues. We then show in

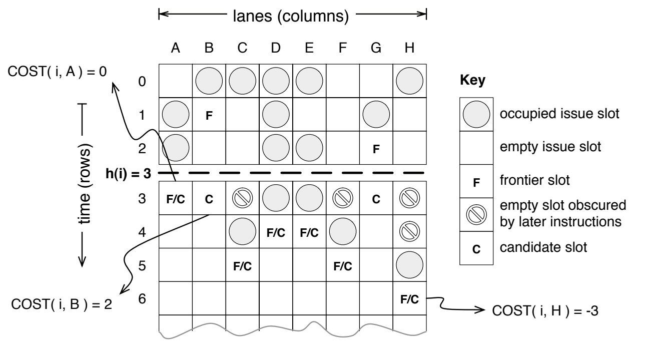


Figure 7. **A scheduling matrix.** The matrix captures the state of a laned machine at each cycle, in this case immediately before instruction i is about to be steered. Rows of the matrix capture the progress of time (time flows downward), and columns represent the machine's lanes. Assuming each lane is a 1-wide machine, each cell in the matrix constitutes a single issue slot.

Section 4.3 that those features render such policies too complex to build. In fact, their operation would amount to a form of out-of-order scheduling, but of a strictly harder variety than is implemented in out-of-order processors.

4.1 Reasoning about steering

As we noted above, our focus in this section is on machines with a modest number of lanes. By this we mean machines whose lane count matches aggregate front-end width, which is smaller than the number of live chains typically active in the window (recall Figure 5). In particular, we restrict our discussion to the 4w–(4×1w) and the 8w–(8×1w) machines. But the results that follow can easily be extended to machines with 2-wide lanes; we do not do so here simply for the sake of expediency.

Our framework for reasoning abstractly about steering is founded on a simple observation: any steering policy induces an *instruction schedule* when it distributes instructions among the lanes, since sending an instruction to a lane implicitly determines the time at which it will execute. The matrix in Figure 7 depicts this idea. It shows the state of a steering-induced schedule just as instruction i is about to be steered.

The line $h(i)$, which we call the (issue) *horizon* for instruction i, represents the earliest possible time at which i will be able to issue. This is a function both of the time at which i's operands will be ready (denoted $data(i)$) and the time at which i is dispatched into the window (denoted $disp(i)$). Specifically, $h(i) = max\{disp(i), data(i)\}$. In addition to its horizon, execution of i is constrained by the current state of the matrix: the most recently occupied issue slot in each column imposes a lower bound on when instruction i will be able to execute at that column. This is of course an artifact of the in-order issue constraint at each lane. We call the first available issue slot at column c the *frontier* of that column, and write $f(c)$ to denote that slot. Each column's frontier is marked with an 'F' in Figure 7.

There are only 8 possible slots into which instruction i can be placed: one in each column. We call these the *candidate slots* for i; each is marked with a 'C' in Figure 7. All of them occur no earlier than both the horizon for i and the frontier of the corresponding column. That is, if we write $cs(i, c)$ for i's candidate slot in column c, then $cs(i, c) = max\{h(i), f(c)\}$.

Different steering choices will have very different repercussions for instruction i and for the ensuing state of the matrix, and hence also for subsequently steered instructions. We can capture those effects by means of a *steering cost* metric:

$$COST(i, c) = h(i) - f(c)$$

When the column to which i is steered is clear (or not important) from context, we will simply write $COST(i)$.

The cost metric will be negative when an instruction becomes ready *before* a column's frontier, in which case $COST(i)$ represents the number of cycles beyond its ready time that instruction i is delayed. The candidate slot in column `H` in Figure 7 is a case in point. We say a steering decision incurs *internal cost* in such cases, thus reflecting the notion that it is the instruction itself that pays the penalty. By contrast, decisions for which the cost metric is positive will be said to incur *external cost*, since these are liable to penalize subsequently steered instructions. A positive cost implies an instruction becomes ready only *after* the column's frontier, so sending it there effectively *hides* a number of issue slots from other instructions. Sending instruction i to column `B`, for example, would incur external cost. The candidate slot in column `A` is an example of a zero-cost steering decision: instruction i is not delayed beyond its horizon, nor is any earlier issue slot wasted by sending it there.

Note that external cost is entirely an artifact of the per-column in-order issue constraint: steering an instruction to an out-of-order core cannot incur external cost because issue slots are not thereby obscured from subsequently steered instructions. We will see shortly that it is precisely the potential for external cost that renders a good policy for laned machines fundamentally hard to obtain.

4.2 Requirements for good steering

We now show that an effective steering policy must pick columns by taking $COST(i)$ into account and, more importantly, that it must do so from *both* an internal (negative) and an external (positive) cost point of view; it is *not sufficient* to take just one of them into account.

4.2.1 Internal cost

We take it as self-evident that paying heed to internal cost is crucial to ensuring good performance.[2] If there is *no* con-

[2]More accurately, internal cost ought to be *minimized* for critical path instructions and *bounded* for non-critical instructions. In both cases, however, internal cost must be taken into account.

straint on how often steering costs are internal, then there will be no constraint on the extent to which instructions are delayed beyond their issue horizons. This is an easy claim to verify. A policy which *randomly* allocates instructions to lanes, for example, places no bound on the potential for negative-cost steering decisions. Our implementation of such a scheme incurs average slowdowns in excess of 70% relative to our monolithic baselines.

The dependence-based policy we evaluated in Section 2.3 is, in fact, an example of a policy that takes internal cost into account. To understand why, it is necessary first to explain in a little more detail how that policy operates. It uses register dependences among instructions to try slot consumers immediately behind the producers of their operands. Specifically, it sends an instruction to a non-empty lane if (one of) its producer(s) immediately precedes it there; if no such lane exists, it sends the instruction to an empty lane, perhaps first stalling until one becomes available. In collocating an instruction directly behind its producer, the policy guarantees that the instruction's issue will be delayed only by that of its producer — a constraint already imposed by dataflow. And in sending instructions to an empty lane, it likewise guarantees that only operand availability will constrain instruction issue. These rules establish a simple invariant: instructions wait in the issue queue only for their operands; the in-order issue constraint imposed by each lane is never exposed, being always subsumed by dataflow constraints. More precisely, if instruction i is steered to lane ℓ, then $COST(i, \ell) \geq 0$. We call this the *internal cost invariant*.

The dependence-based policy's stalling behavior is instrumental to maintaining this invariant. It is through stalling that the policy ensures that instructions are never steered behind independent instructions, and hence that they wait only for their operands once they are dispatched. We can confirm the importance of stalling — and thereby further confirm that internal cost is important — by evaluating schemes in which instructions that would otherwise cause the dependence-based policy to stall are instead steered to a lane picked by some heuristic. This amounts to occasionally slotting an instruction behind independent work. One of the best-performing heuristics we found picks the least-full lane, an indicator that it might drain soon, and therefore that internal cost might be low. It performs substantially worse than the basic dependence-based policy, increasing performance losses by a further 12% on the 4-wide configurations and by more than 35% on the 8-wide machines.

Of course, stalling is not itself benign, since preventing dispatch of one instruction will delay that of younger, perhaps data-ready, instructions yet to be steered. Indeed, we previously diagnosed these stalls as the principal cause of performance problems in a complexity-effective scheduler design closely related to the laned machines [15]. However, a simple experiment suffices to show that stalls are, in fact, just a manifestation of a more fundamental problem. We implemented an oracular version of the dependence-based policy, one that never stalls, but which instead sends an instruction to the same lane that the original policy would otherwise have stalled and waited for. That is, the oracular policy knows ahead of time which lane will drain first. Nevertheless, it performs *no better* than the basic dependence-based policy, which means stalls are hiding a more basic problem with dependence-based steering. That problem is external cost.

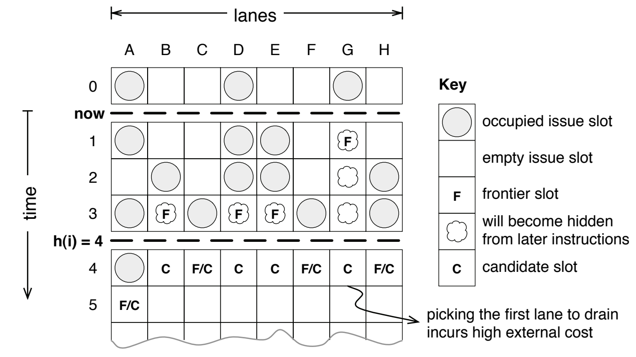


Figure 8. **The problem with dependence-based steering.** At cycle 1, steering logic is about to dispatch instruction i. If i does not depend on a value live at any frontier, it will be sent to column G, which is the one that has just drained. This is precisely the choice with maximum external cost.

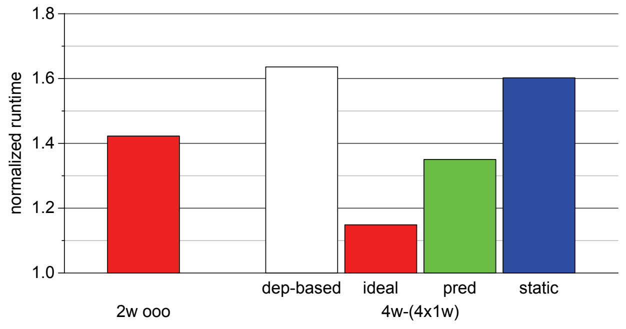
4.2.2 External cost

The dependence-based policy guarantees that its steering costs are zero or positive, never negative. In a sense, it bounds the cost of its decisions *from below*. But it cannot impose any bounds from the other direction: the extent to which steering decisions become largely positive — that is, external — is not controlled. In fact, Figure 8 shows that it is liable to frequently pick the slot that maximizes external cost. When it cannot collocate a consumer with its producer, it picks an empty lane — precisely the one whose frontier is likely to be the furthest away from the consumer's issue horizon (instructions are not always data-ready when they enter the window). Thus, while choosing an empty lane is key to avoiding negative cost decisions, it tends also to push cost into the positive dimension; enforcing the internal cost invariant amplifies external cost.

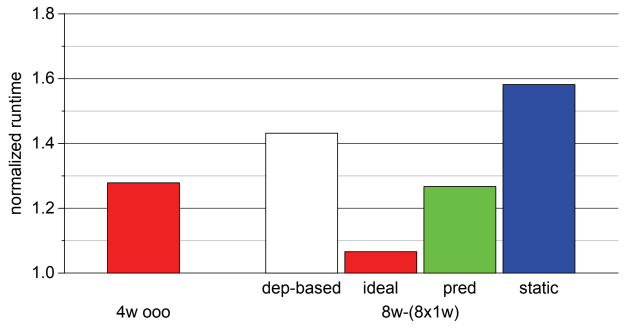
At the root of this problem is the fact that the dependence-based scheme has no means by which to take external cost into account. Dataflow dependences, alone, are not sufficient because bounding the cost of steering decisions *from above* necessarily requires knowing the frontier of each column. And that demands knowing when the last instruction in each lane will issue.

4.2.3 Optimizing both internal and external cost

The previous two sections argued that ignoring either the internal or the external dimension of steering costs is deleterious



(a) 4-wide



(b) 8-wide

Figure 9. **Optimizing both internal and external cost.** The top graph plots runtime of various machine configurations relative to that of our 4-wide monolithic baseline (lower is better); the bottom graph shows performance relative to the 8-wide monolithic baseline. The '2w ooo' and '4w ooo' bars in each graph show, for reference, the performance of a monolithic machine half the size of the baseline. The remaining bars show performance of different steering policies for the laned machine: the idealized cost-optimizing policy ('ideal'); that same policy with oracular knowledge of load latencies replaced with predictions from a load hit/miss predictor ('pred'); and then finally with a static latency prediction in which all loads are assumed to hit in the cache.

to performance. That is, taking both dimensions into account is *necessary* for good performance. We now show that doing so is also *sufficient*. We do so by examining an idealized steering policy in which we equip the steering logic with prescient knowledge of the state of execution at each lane, including exact knowledge of which loads will hit in the cache. In short, we give the steering logic a precise view of the matrix its decisions are inducing. We defer for the moment a discussion of the implementation challenges such a policy would face in practice.

Knowing the state of the matrix permits the idealized policy to directly compute and optimize the cost metric for each of its decisions. For each lane ℓ, it computes $COST(i, \ell)$, then it picks, from among those that have a positive cost (*i.e.* no internal cost), the one whose value lies closest to 0; if no such lane exists, it picks the one whose negative value is closest to 0. Such a policy will tend to optimize $|COST(i)|$ since it tends to avoid slots with extreme internal or external cost. But it gives higher precedence to small positive values than it does to small negative values, meaning it will always prefer penalizing other instructions over penalizing the current one. It is greedy in this respect.

Figure 9 (bars labeled 'ideal') shows this cost-aware policy delivers much better performance than the dependence-based one (bars labeled 'dep-based'), reducing an initial 60% slowdown in the 4-wide configuration to about 15%, and an initial 30% slowdown in the 8-wide machine to about 6%. The 8-wide machine benefits the most because its higher lane count affords the policy more flexibility in optimizing the cost metric. The 4-wide machine offers little opportunity in this regard, especially, for example, when the forward slice of a cache-missing load blocks a lane, thereby consuming 25% of the machine's issue resources.

These performance figures are not the best that an idealized policy can do. Indeed, we have implemented more sophisticated (less greedy) policies in which internal and external cost considerations are weighted differently based on the criticality of each instruction. Although these policies perform even better than the one we present here, we do not show data for them because doing so would add no weight to our main claim, which is that taking both dimensions of the cost metric into account is sufficient for substantially improving performance. Moreover, a more sophisticated heuristic will of course be even more complex to implement; and, as we now show, the greedy policy is already too complex to build.

4.3 Implementing cost-aware steering

Implementing our idealized policy — or, indeed, any policy that takes internal and external costs into account — quickly runs into a number of problems. The culprit, of course, is external cost. As we noted earlier, internal cost can be tackled simply by taking dataflow dependences into account, but targeting external cost, by definition, involves measuring the impact a steering decision will (or might) have on instructions that are not necessarily data dependent on it. Specifically, it demands having some idea of where (in time) each lane's frontier resides, as well as where an instruction's issue horizon lies relative to that. Both of these are inextricably linked to how dataflow in the lanes will be executed. The challenge is knowing and using that information *before* execution actually occurs — when steering decisions are made. This points to the need for an approach similar to the *dataflow prescheduling* techniques previously proposed for wakeup-free schedulers [5, 13]. These maintain information on when each in-flight instruction is expected to execute, and hence when its results will be available.

In terms of our idealized policy, a prescheduling approach would have to operate as follows when steering instruction i; Figure 10 shows the steps graphically.

1. **The issue horizon.** Interrogate in-flight operand state to determine the time at which each of i's source operands will be ready. The horizon, $h(i)$, is the maximum of the current time ($disp(i)$) and the time at which operands will be ready ($data(i)$).

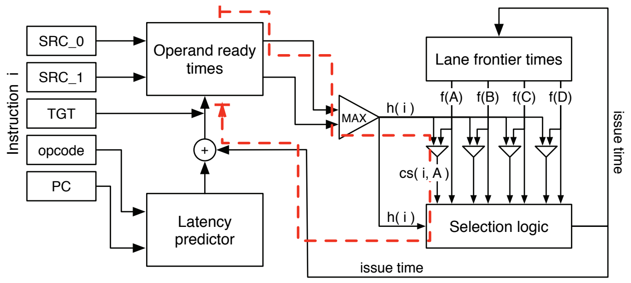


Figure 10. **Cost-aware steering.** The diagram shows the main components involved in making a cost-aware steering decision for instruction i on a laned machine comprising four lanes (named A through D). The selection logic computes $COST(i, \ell) = h(i) - f(\ell)$ for all lanes ℓ, then picks the lane that yields cost closest to 0. The (expected) issue time for i is given by $cs(i, \ell)$, where ℓ is the lane selected for i. This value is used to update the lane frontier times and, together with predicted execution latency, the operand availability time for i's target register. The dashed line running through the diagram identifies the critical loop induced by this feedback.

2. **Candidate slots.** For each lane ℓ, use $h(i)$ and $f(\ell)$ to compute $cs(i, \ell)$, the earliest possible time at which i will be able to issue at ℓ.
3. **Pick a candidate slot.** From among the lanes ℓ for which $COST(i, \ell) \geq 0$, pick the one that minimizes external cost. If no such lanes exist, pick the one that minimizes internal cost.
4. **Update state.** If ℓ was picked at the previous step, update $f(\ell)$ to $cs(i, \ell) + 1$. Also update operand state to record the fact that the output register of i will be available at time $f(\ell) + latency(i)$, where the latter term denotes the expected execution latency of instruction i.

Thus expressed, two problems immediately come to the fore. The first pertains to computation of $latency(i)$ for variable-latency instructions like loads. Returning to Figure 9, the ‘pred’ and ‘static’ bars show that performance rapidly degrades when we compromise a little bit on the accuracy of load latency information, and hence on frontier information. The ‘pred’ bars show the impact of replacing oracular knowledge of load latency with a load hit/miss prediction [17]; those labeled ‘static’ show the effects of assuming all loads hit in the cache. In the former case, both the 4- and 8-wide machines lose most of the performance won by the idealized information; both are now performing at close to the level of the smaller monolithic machine. When static instruction latencies are assumed, performance drops back to the levels of dependence-based steering and, in the case of the 8-wide machine, to even worse levels.

A more fundamental problem with a prescheduling approach is exposed when we consider a superscalar implementation. The problem is that the steering decision for one instruction might depend on the outcome of its predecessors in the same fetch packet. This is analogous to the problems faced by register rename, and indeed by dependence-based scheduling, but now the problem is more acute. Simply put, prescheduling of multiple instructions cannot be performed in parallel, nor pipelined across cycles, because the state that seeds each decision (operand readiness and frontier information) is not available until *after* each steering decision is completed. This is in contrast to register rename logic, which permits back-to-back fetch packets to be renamed over successive cycles because the results of one packet (the new destination tags) are available after a single cycle. The analogous requirement in prescheduling — the destination column for an instruction — is not available until the entire scheduling decision has been completed. The dashed line in Figure 10 delineates this critical loop. In this respect, the prescheduling logic is more complicated than the wakeup-select logic used in out-of-order machines, since the latter does not have to schedule dependent instructions in the same cycle.

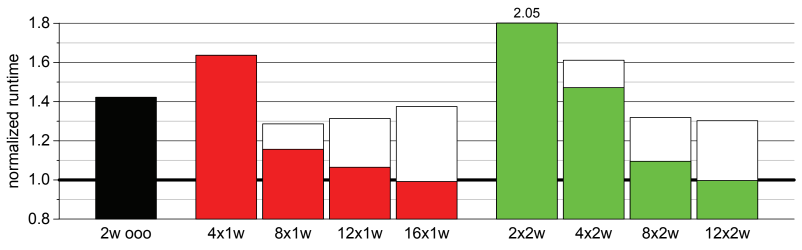
5 Adding more lanes

We saw in Section 3.1 that sustaining an IPC of N generally demands managing more than N live dataflow chains at the same time. Since only the heads of each issue queue are visible to the issue logic, effective management of those chains in a laned machine equates to ensuring that each reaches the head of an issue queue by the time it is ready to issue. We showed in the previous section that doing so is not practicable when there are fewer lanes than there are typically live chains. The only remaining strategy, then, is to forego smart steering and to equip the machine with enough lanes — specifically, enough lane heads — to buffer all the live chains. We show in Section 5.1 that the dependence-based steering policy, which naturally ensures that live chains end up at distinct lanes, benefits as expected from the addition of lanes. But it manages to match monolithic performance only when lane count is high. Moreover, this applies only in the context of our idealized machine model. If we take into account an overhead like global communication, which is inevitably exacerbated by the addition of more lanes, those performance gains are quickly overwhelmed. The picture is not entirely negative, however. We show in Section 5.2 that supporting multiple in-order issue queues per core, together with some modifications to the issue logic, yields almost the same performance as many lanes.

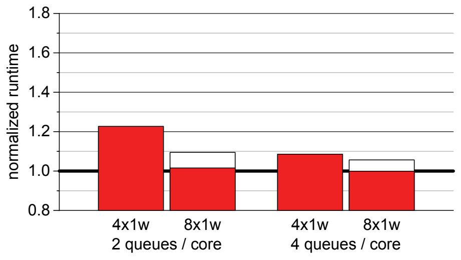
5.1 Performance from more lanes

The dependence-based steering policy slots instructions directly behind a producer, or to an empty lane if that is not possible. Live chains will therefore always end up at distinct lanes, a property which renders dependence-based steering a natural candidate for exploring the potential of a machine with many lanes. Figure 11(a) shows its performance on machines with a 4-wide front-end and 4 or more lanes for instruction execution.[3] Performance losses relative to the 4-wide monolithic machine drop below 10% only at the 12×1w

[3]For lack of space, we omit data on machines with an 8-wide front-end. Their performance trends are very similar to those in Figure 11, however.



(a) Multiple cones



(b) Multiple issue queues per core

Figure 11. **More lanes help performance.** In the graph on the left, each bar shows harmonic mean runtime (lower is better) of various laned machines, all with a 4-wide front-end, relative to our 4-wide monolithic machine. For reference, the '2w ooo' bar shows the performance of our 2-wide monolithic machine. The shaded portion of each bar shows performance in the absence of penalties introduced by fusion. The unshaded portion shows additional slowdowns that would be incurred if a global communication penalty is imposed: we add 2 cycles for every 4 lanes we add to the narrowest (leftmost) configuration. The graph on the right shows the effect of having more than one in-order issue queue at each core. Each pair of bars shows the results of fusing 4 and 8 1-wide cores, first with 2 issue queues per core, then with 4 per core. The unshaded portion of the bars shows the effect of a 2-cycle global communication penalty on the 8-core configurations.

and the 8$\times$2w configurations. In other words, we would need to have more than twice as many lanes as front-end width before performance of a 4-wide monolithic machine is matched. Note that these results are consistent with the data in Figure 5, which showed that covering more than 90% of the live chain distribution would demand 14 or more lanes.

The above results are optimistic, of course. At a minimum, they assume that inter-lane (global) communication is free. While one *could* argue that four 1-wide machines could be floorplanned to perform inter-operation forwarding as efficiently as a 4-wide monolithic machine, it is hard to believe the same holds for eight or more cores. The unshaded portions of the bars in Figure 11(a) show the relative execution time when global communication penalties are introduced, 2 cycles for every 4 additional lanes beyond the fetch width of the machine. With a two-cycle forwarding latency, performance of the 4w–(8$\times$1w) machine now lags almost 30% behind that of a 4-wide monolithic processor; and modest increases in that latency very quickly counter the benefits of adding more lanes. Because a larger number of lanes exacerbates other complexities of building a laned machine — among them, the crossbar for steering fetched instructions to lanes, the distributed data cache, and static and dynamic power consumption — it is difficult to build a compelling case for such machines.

5.2 Multiple lanes per core

The benefit of many lanes derives ultimately from the availability of issue queue slots for live chains, not from the extra issue bandwidth. One potentially appealing design point, therefore, is to share issue resources among 2 or more issue queues. This would provide steering logic with a sufficient number of target issue queues and, at the same time, would mitigate the problems of fusing too many cores. Moreover, such an approach might combine well with *multithreading* capabilities in each core, which already demand buffering capabilities for more than one instruction stream. But issue logic would have to evolve from a simple in-order scheme to one capable of selecting ready instructions from among more than one issue queue. A natural candidate for such a scheme is *dependence-based scheduling* [14], which picks ready instructions from among the heads of FIFO buffers. It is, in a sense, a very small dynamic-scheduler, since the heads of each FIFO buffer together constitute an out-of-order issue queue. It is needed to achieve slip among the local issue queues, slip that previously arose naturally through the decoupled nature of execution at each lane.

Figure 11(b) shows the effects of adding a dependence-based scheduler to our 1-wide in-order cores, each now equipped with more than one issue queue. Performance remains very close to that of buffer-equivalent laned machines. The 4$\times$1w machine, for example, when equipped with 2 FIFO buffers per core, almost matches the basic 8$\times$1w machine; and 4 FIFO buffers brings it close to the original 16$\times$1w machine's performance. These results do not conclusively prove that slip-oriented execution is a viable alternative to conventional dynamic scheduling, but they do show that a modest number of cores, together with reasonably modest enhancements to the issue logic, can mimic the benefits of many lanes. It remains to be seen whether those changes leave the design simple enough to be more compelling than, say, fused machines that use small out-of-order cores as the unit of composition [6].

6 Conclusion

The idea that in-order cores might be fused, on demand, into larger out-of-order processors is an extremely compelling one. It would offer chip vendors an opportunity to design a simple and power-efficient in-order substrate from which aggressive ILP processing capability can be dynamically synthesized when, and if, it is needed. Such a chip would cater to a wide variety of workloads, potentially serving all of them better than would designs such as heterogeneous CMPs, whose rigid partitioning is bound to fall outside the requirements of some workloads.

We explored the prospects of such designs by evaluating the performance potential of *slip-oriented out-of-order execution*, the underlying execution model that characterizes these machines. We find that basic properties of dynamic dataflow fundamentally constrain that model. Specifically, matching the performance of conventional out-of-order execution requires the ability to monitor, and ensure prompt execution for, many simultaneously-active chains of dataflow, many more than the average ILP extracted from the program. As a result, modest designs, which fuse a reasonable number of cores, and which employ implementable instruction steering schemes, exhibit best-case IPC performance well below that of smaller out-of-order machines — designs we are able to build, at modest complexity, already. It is, in principle, possible to do better with more sophisticated steering policies, but the complexity thereby introduced turns out to be no better than conventional out-of-order issue logic; indeed, it appears to be worse. Though configurations with a large number of cores have better performance potential, the overheads introduced by fusing so many cores would render them impractical. That said, the ability to manage multiple (in-order) issue queues at each core, which requires some modifications to the issue logic, might be a compelling compromise between in-order execution and full out-of-order capabilities at each core. The ultimate appeal of such an approach depends on trade-offs between performance, power and design complexity, all of which require the exploration of more specific designs than the broad class we examined here.

Acknowledgments

This research was supported in part by NSF CCF-0429561, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. The authors would also like to thank Lee Baugh, Andy Glew, Kshitiz Malik, Naveen Neelakantam, Amir Roth and Sam Stone for their many useful comments.

References

[1] AGEIA Technologies, Inc. PhysX by ageia. `http://www.ageia.com/pdf/ds_product_overview.pdf`.

[2] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 337–347, December 2000.

[3] R. Canal, J.-M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, pages 133–142, January 2000.

[4] ClearSpeed Technology. Whitepaper: CSX processor architecture. `http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf`, February 2007.

[5] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 253–262, June 2003.

[6] E. Ipek, M. Kirman, N. Kirman, and J. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, June 2007.

[7] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.

[8] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. Keckler. Composable lightweight processors. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 381–393, December 2007.

[9] 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.

[10] 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.

[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, December 2003.

[12] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[13] 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.

[14] 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.

[15] P. Salverda and C. Zilles. Dependence-based scheduling revisited: A tale of two baselines. In *6th Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2007.

[16] Sun Microsystems. Throughput computing: Changing the economics and ecology of the data center with innovative SPARC technology. `http://www.sun.com/processors/whitepapers/throughput_whitepaper.pdf`, November 2005.

[17] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.

[18] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Conference on High Performance Computer Architecture*, pages 25–36, February 2007.