A Criticality Analysis of Clustering in Superscalar Processors

Pierre Salverda Craig Zilles

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

Abstract

Clustered machines partition hardware resources to circumvent the cycle time penalties incurred by large, monolithic structures. This partitioning introduces a long inter-cluster forwarding latency and the potential for load imbalance, both of which degrade IPC and thus counter the cycle time benefits of clustering.

We show that program dataflow can be mapped to clustered machines so as to achieve an IPC rivaling that of an equivalent monolithic machine. That is, the IPC penalties observed by extant schemes are largely an artifact of instruction steering and scheduling policies. Using critical path analysis, we investigate and uncover the main causes for this performance loss. By way of code samples, we illustrate those causes and propose three policies for mitigating them. First, we introduce a new metric, likelihood of criticality, and show how it can halve the performance lost to contention-induced stalls. Second, we develop a stall-over-steer policy that addresses performance lost to inter-cluster forwarding delay. Finally, we show that a proactive load-balancing policy is necessary to improve the distribution of ready instructions among the clusters. Together, these three policies yield performance on 2-, 4- and 8-cluster implementations of an 8-wide machine that is within 2, 4, and 6%, respectively, of the monolithic equivalent.

1 Introduction

Out-of-order superscalar execution is a compelling means for exploiting both instruction- and memory-level parallelism. The former is necessary for boosting performance when fine-grained parallelism exists in the instruction stream; the latter for hiding increasing memory latencies, especially in programs that are memory-bound. Performance in this execution model is improved by increasing the machine's issue width and window size. A larger issue width increases the number of instructions executed per cycle, which improves the machine's ability to exploit parallelism when it is available in the instruction stream. A larger window increases the out-of-order potential of the machine, which is important for hiding long latency memory operations. Unfortunately, naively scaling issue width and window size significantly constrains the achievable clock frequency [19].

Clustered microarchitectures are a natural solution to these scalability problems. By replicating structures, a clustered architecture can achieve a large *aggregate* issue width and window size. The Alpha 21264 [14], with 2 execution clusters and a replicated physical register file at each, is perhaps the best-known example of this approach. Clustered machines are appealing because they can be scaled by making structures small and more numerous rather than larger and monolithic. Many of the power-, clock- and complexity-related problems encountered by monolithic designs are thereby circumvented.

Of course, these benefits do not come without cost. Clustered machines are prone to degraded performance, the two principal causes of which are *inter-cluster communication delays* and *increased contention for diminished resources* at each cluster. The former can increase the effective latency of instructions by increasing the time it takes to forward their results to their consumers. The latter effect arises because of the potential for load imbalance, which reduces the effective aggregate window size and issue width of the machine. These factors give rise to two conflicting objectives in the design of a clustered machine: achieving *locality* to hide the communication delay from producer-consumer pairs versus achieving *load balance* to avoid contention-induced stalls at each cluster.

While the trade-off between locality and load balance lies at the heart of the performance challenge, understanding the problem purely in terms of this trade-off is too simplistic. In the control-intensive programs in which we are interested, not all instructions contribute equally to performance, so attempting to achieve good locality and load balance, *on average*, is somewhat misdirected. Indeed, there will be cases where many dependent instructions are not collocated and load is far from balanced, but where performance is optimal because the few important (critical) instructions are not penalized.

That criticality is important is, of course, nothing new — it plays a key role in statically-scheduled clustered architectures, the earliest examples of which date back to the VLIW machines developed in the early 1980s. Those machines expose clustering at the architectural level so that it can be considered in the compiler's scheduling pass [6, 16]. Within a given scheduling scope, the compiler knows the dataflow height and width of each computation, enabling it to generate near-optimal schedules by using an integrated cluster-assignment and scheduling pass [12, 18]. Dataflow height serves well as a measure of criticality in this framework because of the static nature of the underlying machine. In contrast, dataflow height is an impractical and imperfect criticality metric in dynamic machines. It is impractical because it requires backward dataflow knowledge and imperfect because execution can be overlapped with branch resolution and data cache misses.

In this paper, we focus on clustering in dynamically-scheduled machines, like the one depicted in Figure 1. These machines differ from their static counterparts in two key respects. First, cluster assignment and instruction scheduling — two steps that are unified and implemented by the compiler in the static machines — are now

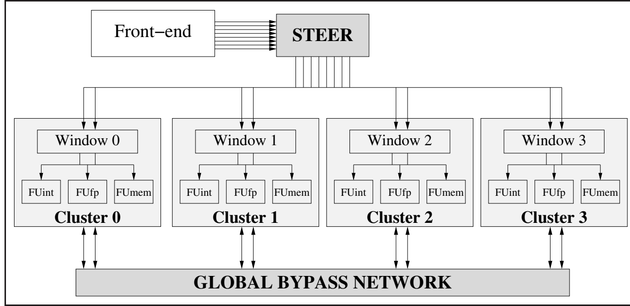


Figure 1. A clustered microarchitecture. The machine comprises a monolithic front-end and a partitioned execution core. In this case, an 8-wide issue and execute bandwidth is partitioned equally among 4 clusters. Cluster assignment is performed by the *instruction steering* logic. Each cluster is a self-contained dynamically scheduled execution core: instructions arriving from steering enter the scheduling window, from which they will issue when their operands become available. Those operands will either have been produced locally or will be received — after the global communication latency has lapsed — from a remote producer via the global bypass network.

decoupled and are performed, dynamically, in hardware. Second, scheduling itself is now *distributed* because the window of in-flight instructions is partitioned. Together, these differences introduce a fundamentally different set of challenges for cluster assignment and instruction scheduling.

Despite a very large body of research in this area, there remain a number of unanswered questions about the basic characteristics of the design space. For example, it is not clear what portion of the observed performance penalties, even in the best results seen so far, are inherent to clustering as opposed to merely being artifacts of sub-optimal steering and scheduling policies. Moreover, if sub-optimal policies are to blame, it is not even clear to what causes their inefficiencies can be ascribed; there is at present no characterization of the causes, nor a quantification of the extent to which each contributes to performance loss. These are the questions that motivate this work. Accordingly, and in contrast to much previous work, our objective is not to develop *mechanisms* for achieving better performance for a specific microarchitecture. Rather, our aim is to discover the factors that underly performance penalties in a spectrum of microarchitectures, and, in so doing, to define and evaluate *policies* (*not* mechanisms) that are likely to alleviate them. Specifically, our paper makes the following contributions.

- **Potential.** In an idealized study (Section 2), we prove the existence of cluster assignments and instruction schedules that yield performance almost identical to (within 2% of) the performance of an equivalent monolithic machine. That is, we demonstrate that penalties observed by extant schemes are not inherent to the underlying hardware, but are entirely an artifact of the policies used to manage it.
- **Attributing lost cycles.** We perform a criticality-based analysis of a "state of the art" steering and scheduling policy to identify the causes of this loss of performance (Section 3). We identify two main culprits: contention among known-critical instructions and load-balancing of critical instructions.
- **Policy.** To mitigate these effects, we present three policies to improve steering and scheduling.
  – *Likelihood of criticality-based scheduling.* We introduce the likelihood of criticality (LoC) metric to extend Fields *et al*'s binary notion of criticality [10] to a more continuous spectrum (Section 4). This permits a criticality-based scheduler to prioritize among critical instructions more intelligently.
  – *Stall over steer.* We show that, in execute-critical regions of code, it is better to stall steering when a cluster is full than steer critical instructions away from their producers (Section 5). The LoC metric can be used to guide this selective stalling.
  – *Proactive load-balancing.* On machines with narrow clusters, we show that load-balancing should be performed so as to steer all but the most critical consumer to other clusters (Section 6). Because the most critical consumer is often *not* the first consumer, this requires learning which early consumers should be proactively steered away to make room for a subsequent, more critical, consumer.

For each of these policies, we provide code examples to give the reader an intuition why they are necessary. In Section 7, we complement these anecdotal examples with performance results from a timing simulator, thereby also demonstrating the effectiveness of each of the policies. We show reductions in the penalty due to clustering of one-half to two-thirds for 2-, 4-, and 8-cluster machines, which brings the slowdown with respect to our idealized study (Section 2) to less than 5% for all machines. We conclude in Section 8 with a summary of our contributions and a brief discussion of our plans for future work, where we aim to develop realistic mechanisms for implementing our various policies.

2 The potential of clustering

Research into clustered designs has tended to focus on implementation mechanisms and comparative studies of various steering policies [2–5, 7, 10, 13, 15, 19, 20, 23]. While these are of course important issues, focusing exclusively on them leaves open the fundamental question of how well those mechanisms and policies exploit the underlying hardware to its fullest. We feel this is an important subject to address, for two reasons. First, it underpins the tenability of clustered microarchitectures as a solution to the scalability problems alluded to earlier. If performance penalties are inherent to the clustered approach, and those penalties are as large as they presently appear to be, the whole design space becomes less attractive. Second, even if certain penalties are inherent, it is important to know what they are if we are to gauge the success of different proposals, and hence justify effort being directed at finding better solutions.

Our objectives in this section are twofold. First, we use an idealized study to show that clustered microarchitectures are capable of achieving performance that is almost identical to monolithic machines. That is, partitioning the window and execution resources does not, of itself, impose any significant performance penalty. Second, we show that there is a significant gap between the hardware's performance potential and what a "state of the art" steering and scheduling scheme extracts from it. We conclude from this evaluation that clustering is indeed an attractive design option — if we can

develop the right policies to manage the hardware, then the benefits that clustering offers to clock cycle time, power and complexity need not be won at any (significant) cost to IPC. These observations serve as motivation for an in-depth investigation into the causes behind performance loss in the existing schemes, a study to which we devote the remainder of the paper.

Before we present any data, we describe the architectures and the simulation infrastructure used throughout our work. We want to emphasize up front that the specific architectural parameters we choose are not particularly important — we are interested in the *fundamental* trade-offs that underly clustering in general, not in the specifics of how to make a given machine run faster.

Front-end	8-wide, 13 stages to dispatch. Perfect instruction cache. gshare branch predictor with 16 bits of global history.
Issue	128-entry scheduling window, 256-entry ROB.
Execute	Up to 8 instructions per clock, any mix of up to 8 integer instructions, 4 floating point and 4 memory instructions (load or store). Instruction latencies match the Alpha 21264 [14] (*e.g.*, 3 cycle load-to-use). Perfect memory disambiguation.
Memory	32KB 4-way set associative L1 cache, 2 cycle access time. 20-cycle, infinite L2 cache.

Table 1: **Baseline (monolithic) machine parameters.**

2.1 Methodology

Throughout this paper, we measure the effectiveness of a clustered machine in terms of an equivalent monolithic machine. We choose an 8-wide, out-of-order superscalar to serve as that baseline. The clustered machines are configured so as to apportion the monolithic machine's execution resources equally among the clusters. We therefore examine three cluster configurations: two 4-wide clusters, four 2-wide clusters and eight 1-wide clusters, henceforth referred to as the 2x4w, 4x2w and 8x1w configurations, respectively; the baseline will often be referred to as the 1x8w configuration.

As a vehicle for evaluating these architectures, we make use of a trace-driven timing simulator. Table 1 summarizes the various architectural parameters for the simulator's 1x8w configuration. The clustered configurations are identical in all respects, except that, as noted, the execution resources are divided among the clusters. For example, the 4x2w configuration, which is depicted in Figure 1, divides the 128-entry scheduling window into four 32-entry windows at each cluster. Likewise, the 8-wide execution bandwidth is divided equally among the four clusters, so that each can issue 2 instructions per clock (no more than 2 integer operations, 1 floating point operation and 1 memory operation per cycle).[1] All clusters load from/store to a shared L1 data cache.

In any clustered architecture, inter-cluster latency and bandwidth are important design parameters. In our experiments, we modeled latencies from 1 to 4 cycles, but, due to space constraints, show results for a 2-cycle latency only. Although the absolute performance figures differ for the various latencies, the trends we observe, and the conclusions we draw from them, do not. Indeed, the observations we make relate primarily to properties of program dataflow, which are affected, first and foremost, by the number and width of the clusters, not by the latency and bandwidth among them. In terms of bandwidth, we assume the global bypass network has enough capacity to support peak execution rates, so we do not model contention for communication slots. Nonetheless, we do monitor inter-cluster communication and find that, on average, our policies incur 0.12, 0.2 and 0.25 global values per instruction for the 2-, 4- and 8-wide configurations, in all cases slightly less than the baseline steering policy we compare our schemes against (described in more detail later). A more detailed analysis of the effects of a limited-bandwidth interconnect is beyond the scope of this paper.

We simulate an infinite 20-cycle L2 cache in order to reduce simulation times and cache warm-up times. We verified that these experiments have very similar CPI breakdown (as we present in Section 3) to runs that use a finite L2 cache and a 200 cycle memory, except for a somewhat smaller CPI contribution from memory. Thus, our results may (conservatively) overestimate the negative impact of clustering.

Since we are specifically interested in non-numeric programs, we use the Spec 2000 integer benchmarks in all our simulations. Benchmarks are compiled using the DEC C Alpha compiler (V5.9-005), with peak optimization, but no profile feedback. For each benchmark, we average results from three 100 million instruction runs (after warming up the branch predictor and cache) starting at 3, 5 and 8 billion instructions into the run.

2.2 An idealized study

To explore the performance potential inherent to clustered hardware configurations, we examine traces of instructions retiring from the 1x8w machine's back-end. Using a list scheduler configured for each of our cluster configurations, we then build schedules for the trace.[2] We use the term *schedule* here to denote a *placement* (at a cluster) and a *slotting* (into an issue slot) of each instruction in the trace. That is, our list scheduler performs both steering and instruction scheduling in a single pass. The scheduler adheres to the per-cycle issue constraints imposed by the underlying hardware being modeled, and also imposes the global communication penalty when values are communicated between clusters. We are also true to the fetch constraints imposed by the 1x8w machine's front-end: an instruction cannot be scheduled earlier than the time it was dispatched into the machine's out-of-order window, and the latency of branch mispredictions is observed.

The analysis we perform is idealized in two main respects. First, the scheduler has a global (monolithic) view of all in-flight instructions and thus treats only the functional units themselves as clustered. This is deliberate — we are interested in the basic capabilities of the underlying hardware, not in the penalties introduced by policies used to manage it. Second, the scheduler has exact future knowledge because it sees all instructions in a region at once. It uses this information to prioritize instructions, giving precedence to those from which long dataflow chains emanate and to those that

[1] We round up partial resources, so each cluster in the 8x1w machine has a memory port and a floating point ALU.

[2] We schedule the whole execution trace by dividing it into regions that are separated by instructions — mispredicted branches, in particular — on the critical path. By summing the spans of all schedules thus obtained, we get a *conservative* estimate of overall runtime.

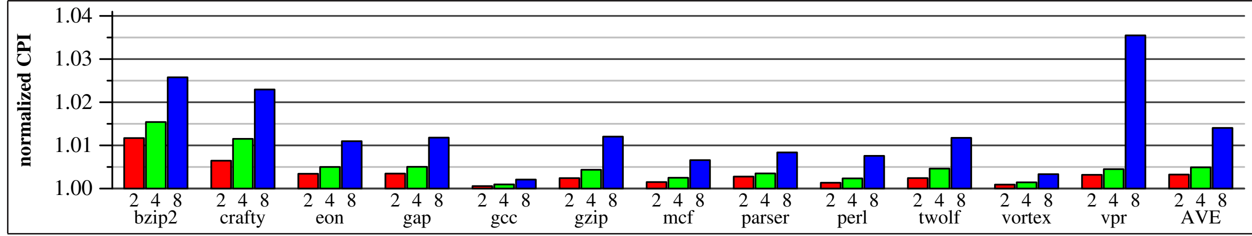


Figure 2. Idealized list scheduling. Bars labeled '2', '4' and '8' denote the 2x4w, 4x2w and 8x1w clustered configurations, respectively. CPI is normalized to that of the 1x8w configuration.

reside on the backward dataflow slice of mispredicted branches. The scheduler takes locality into account by trying to collocate consumers with their producers. When coupled with the depth-based priority scheme, this tends to keep long, likely-critical, dataflow chains collocated, and thus free from global delays.

Figure 2 plots the schedule times we obtained using the aforementioned approach. We make two observations about the data. First, all clustered configurations achieve average performance that is less than 2% slower than the 1x8w configuration.[3] It appears that partitioning the underlying hardware does not, of itself, impose any fundamental limits on how close the machine's IPC can approach that of the monolithic equivalent.[4] This result permits us to claim, in Section 2.3, that state of the art schemes for assigning instructions to clusters are significantly underperforming.

A second observation relates to benchmarks `bzip2`, `crafty` and `vpr`, for which performance, particularly on the 8x1w configuration, stands in contrast to the other benchmarks. From our critical path analysis (discussed in Section 3), we found that the bulk of the discrepancy — about 70% of extra cycles in the benchmarks overall and over 80% in the three worst-performing benchmarks — can be attributed to a single cause: *convergent dataflow*. This problem arises when dyadic instructions consume values on dataflow edges that both have little or no slack. To a large extent, convergent dataflow imposes fundamental, though small, limits on the performance potential of a clustered machine.

In the case of `bzip2` and, to a smaller degree, in `crafty`, convergence incurs penalties via global communication. Figure 3 shows why. For the 8x1w configuration, the list scheduler is assigning instructions exactly as indicated in Figure 3b, and is therefore performing optimally on that code; convergence in this case imposes a fundamental limit on the performance potential of the 1-wide configuration. In the 4x2w and 2x4w configurations, however, the scheduler does not always collocate the producers that precede convergence, so it occasionally incurs a global penalty in both cases; hence the relatively poor performance shown by the scheduler on these configurations as well. Conceivably, the scheduler could be enhanced to deal more effectively with these scenarios.

Convergence in `vpr` incurs contention stalls. In this case, the convergence occurs in dataflow "hammocks", where a single instruction produces a value for two chains of consumers (diverging dataflow), which, in turn, subsequently converge at a dyadic consumer. When this occurs, the scheduler attempts to collocate both consuming chains with the single producer. On the 8x1w configuration, this incurs a serialization of otherwise parallel work, which manifests itself as resource contention. For the 8-cluster machine, these slowdowns are unavoidable — either contention stalls will occur or global penalties must be incurred. In contrast, the 4x2w and 2x4w configurations, which admit some parallelism intra-cluster, do not suffer from this problem.

[3]These results are remarkably stable across the various inter-cluster latencies we modeled. For example, with a 4 cycle global penalty, average performance loss for the 2x4w and 4x2w configurations is still below 2%; the 8x1w configuration loss degrades to a little over 4%.

[4]Instruction replication, which has been advocated for statically-scheduled clustered machines [1, 17], therefore does not appear to be *necessary* for dynamic machines.

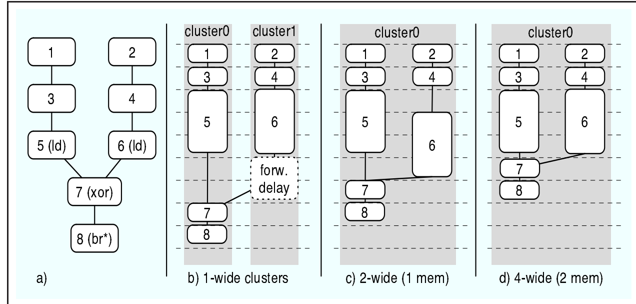


Figure 3. An example of convergent dataflow in `bzip2`. (a) Dataflow leading into a mispredicted branch; nodes are labeled with their fetch order and their operation type (unlabeled instructions are single-cycle integer operations). **(b)** The optimal allocation for 1-wide clusters incurs one forwarding delay; there would be 3 cycles of contention if it was assigned to one cluster. **(c)** With 2-wide clusters with a single memory port there is a single cycle of contention for the memory port. **(d)** With clusters that have 2 memory ports this code could execute at full speed.

In general, convergent dataflow poses a difficult problem for clustered machines because dealing with it requires advance knowledge that convergence is imminent. Even with forward knowledge, which our idealized scheduler has, there is no single policy that caters for all scenarios — there are cases where collocation is better (*e.g.* `bzip2`-like convergence on the 4x2w configurations) and cases where load-balancing is potentially better (*e.g.* large hammocks on the 8x1w configurations). For the 8x1w configuration, in particular, the problem is a fundamental one because the demand for parallelism necessarily incurs either global communication penalties or contention stalls. That said, the overall effect is still small — the 8x1w configuration is never more than 4% slower than the monolithic schedule. Clearly, critical dataflow in these benchmarks seldom demands support for intra-cluster parallelism; most parallelism can be exploited *inter*-cluster.

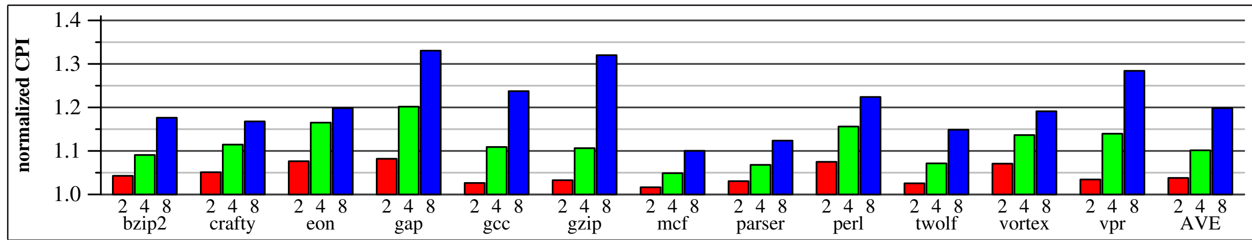


Figure 4. Focused steering and scheduling. Like Figure 2, bars labeled 2, 4 and 8 denote the 2-, 4- and 8-cluster machines. In contrast to that figure, however, the scale on the y-axis is now larger by an order of magnitude.

2.3 The state of the art

Having shown that clustered architectures are capable of monolithic performance, we now consider the performance that extant schemes are achieving. For this purpose, we chose the *focused steering and scheduling* policy proposed by Fields *et al* [10]. This is representative of the state of the art in terms of policies for managing dynamic clustered machines. It serves, therefore, as a good basis for quantifying the difference between what hardware can achieve in theory and what current policies are extracting from it in practice.

The focused steering and scheduling scheme equips the pipeline with a criticality detector that samples the retiring instruction stream, looking for instructions whose execution resides on the program's critical path. Results from the detector feed a criticality predictor, which is a PC-indexed table of saturating counters. The counter for a given PC is incremented when the corresponding instruction is detected as critical, and decremented otherwise; when that value exceeds a certain threshold, the instruction is classified as critical. The machine uses a dependence-based steering policy [13], modified to use output from the criticality predictor: whenever there is a choice of cluster to which a consumer can be sent, the one holding the critical producer is given preference. In addition, instruction scheduling logic is modified to give priority to critical instructions, the objective being to reduce the exposure of critical instructions to contention-induced stalls.

We have incorporated the Fields critical-path detection logic into our simulation environment and implemented the focused steering and scheduling policies. The performance results we obtain, which are in general agreement with those published by Fields *et al*, are plotted in Figure 4. Whereas performance of the 2x4w configuration is usually within 5% of the monolithic machine, the 4x2w configuration shows several slowdowns in excess of 10%; the 8x1w configuration averages a slowdown of 20%. Overall, these figures correspond to an order of magnitude increase in the penalties incurred by the list scheduler (Figure 2). We devote the remainder of the paper to an analysis of the effects that underly this discrepancy.

3 Analysis of the lost cycles

In light of the data presented in the previous section, an obvious question arises: "Why do extant steering and scheduling policies significantly underperform?" While it is clear that forwarding delay and load balance underly the performance problems, aggregate statistics relating to these metrics are not meaningful — many instructions can incur these penalties without impacting the overall execution time [9]. That is, we are only interested in cases where forwarding delay and contention stalls contribute directly to the program's execution time.

To attribute runtime cycles lost to these penalties, we use the critical path model developed by Fields *et al* [10]. This involves post-processing an execution trace to find a chain of dependences — including constraints imposed by fetch, dataflow and commit — that together determine total runtime. Having thus delineated a critical path, we then attribute cycles to inter-cluster forwarding and resource contention as follows. Each time critical producer-consumer dataflow crosses clusters, we attribute 2 cycles to forwarding delay. Contention cycles are attributed by adding all critical execute cycles not accounted for by functional unit latency, forwarding delay, and memory latency.

It should be noted that our attributions are not always unique. Previous work has demonstrated the presence of parallel critical and near-critical paths [8]. Thus, a performance improvement is not guaranteed if slowdowns on only one critical path are addressed. Nevertheless, we find this methodology is accurate enough to provide insight into the behavior of clustered machines.

Figure 5 shows the composition of the critical path for the focused steering and scheduling policy (same data as Figure 4). The cycle breakdown shows that, even with dependence-based steering and focused scheduling, the critical path frequently suffers from clustering-induced penalties: it often crosses clusters (forwarding latency) and it often stalls because critical instructions are not being executed as soon as they are ready (contention). On occasion, the execution time that we attribute to these two effects exceeds the slowdown relative to the monolithic machine. As noted earlier, this does not suggest that we will outperform the monolithic machine if we eliminate these stalls. Rather, this is indicative of a shift in the critical path — by introducing stalls, the clustering has forced what was previously a near-critical path to become critical.

Much of the observed change in the critical path is a shift from *fetch criticality* to *execute criticality* [10]. This occurs because the back end is no longer keeping up with the front end, causing the window to fill quicker than it drains. Instructions are, as a result, more likely to be put into the window before they are data ready, making them execute-critical. While this is evidence that clustering is having an impact, it does not provide insight into why clustering-induced stalls remain, despite the criticality-based steering and scheduling policies.

To answer that question, we can zoom in on the critical path to find the main contributors to contention and forwarding cycles. Figure 6 summarizes the results of that analysis. For contention-related stalls (Figure 6(a)), we find that as much as two-thirds come from instructions delayed *in spite of being correctly predicted as*

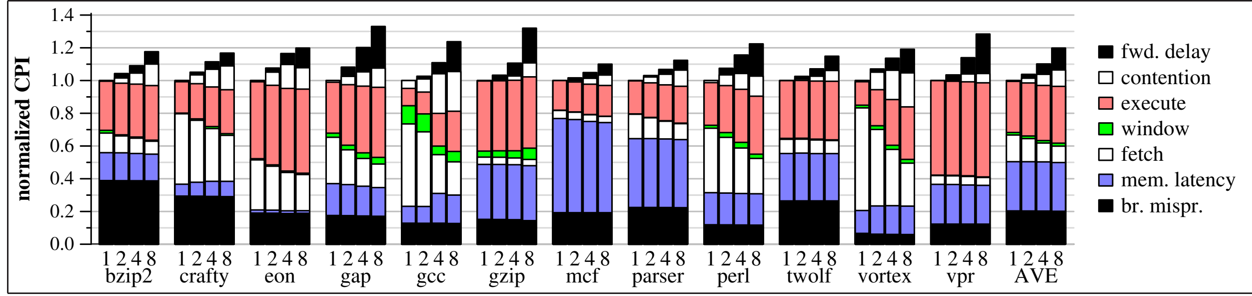
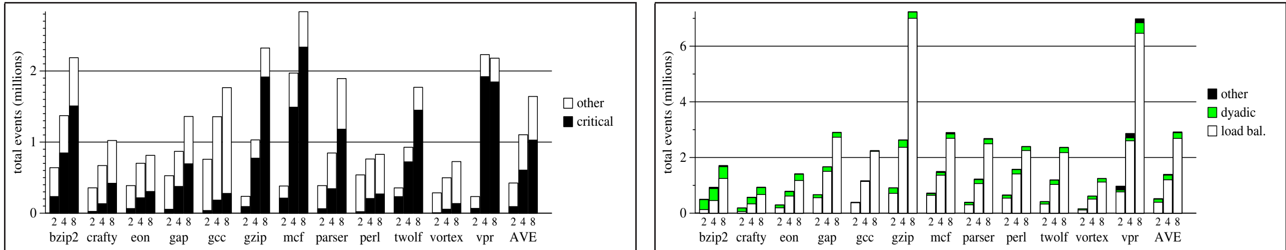


Figure 5. Critical path breakdown for monolithic and 2-, 4-, and 8-cluster machines using focused steering and scheduling. All results are normalized to the performance of a monolithic machine (labeled **1**).



(a) **Contention stalls.**

(b) **Forwarding delay.**

Figure 6. Where the lost cycles went. **(a)** Contention stalls among critical instructions are incurred predominantly by instructions that have been predicted critical. **(b)** Forwarding delay is incurred by critical instructions mainly as a result of load-balance steering; only in `bzip2` and `crafty`, which have an abundance of convergent dataflow, do dyadics dominate.

critical.[5] We will show in the next section that such stalls occur because multiple predicted-critical instructions are contending for the same resources. In terms of forwarding cycles (Figure 6(b)), we find that the dominant cause is load-balance steering. This occurs when an instruction's critical source operand will be produced by a cluster that is currently full; the consumer is in this case assigned to the cluster with the fewest in-flight instructions.

In the sections that follow, we demonstrate these effects by way of code examples and then discuss changes in policies that can address them. First, in Section 4, we show that the contention problem arises from Fields's binary notion of criticality. What is needed is a more continuous spectrum that will allow us to distinguish between, and hence prioritize among, critical instructions. We introduce the notion of *likelihood of criticality* for this purpose. In Section 5, we then demonstrate that load balancing is exactly the wrong thing to do when code is execute-critical; it is preferable to *stall* instruction steering when the desired cluster is full. We also show that likelihood of criticality is a good metric for discerning execute-critical regions of code, and hence for driving the decision to stall rather than steer. Finally, we show in Section 6 that, as a positive side effect of stalling on critical nodes, we achieve an improved distribution of ready instructions. However, we find that a number of factors limit the extent to which this benefit can be felt. To alleviate that problem, we need *proactive load-balancing* to push non-critical consumers away from their producers.

[5] In other words, contention does *not* arise because the criticality predictor produces false negatives. Instructions that are truly critical are correctly predicted as such.

4 Likelihood of criticality

The effect that underlies the contention stalls incurred by predicted-critical instructions is most easily demonstrated with an example. Figure 7 shows a loop from the benchmark `vpr`. The dataflow exhibits a *spine and ribs* structure, a common feature in programs. The dominant spine, which flows through the instruction labeled **b**, computes a loop-carried dependence. Dataflow periodically diverges from this spine to feed the ribs, which terminate on stores and branches; the rib that starts with the instruction labeled **a** includes a hard-to-predict branch.

The contention problem arises between instructions **a** and **b**. Both of these are frequently predicted as critical because both are on the backward slice of the (critical) mispredicted branch. Since both instructions consume from the same source register, they are both routed by the dependence-based steering policy to the same cluster. Being both ready to execute at the same time, they will contend for an issue slot on a machine with 1-wide clusters. The scheduler, which sees both as critical, breaks ties by choosing the older instruction — in this case, instruction **a**. This is the wrong choice for every iteration but the last, because only in the last iteration is instruction **a** actually critical; on all other iterations, instruction **b** (the truly critical one) incurs a contention stall.

This problem arises because of the binary nature of the criticality predictions. If we instead predict *how likely an instruction is to be critical* — **b** is much more likely to be critical than **a** — and prioritize instructions based on that likelihood, we can achieve a better schedule than with a prediction of critical/not-critical.

To address this problem, we introduce the *likelihood of critical-*

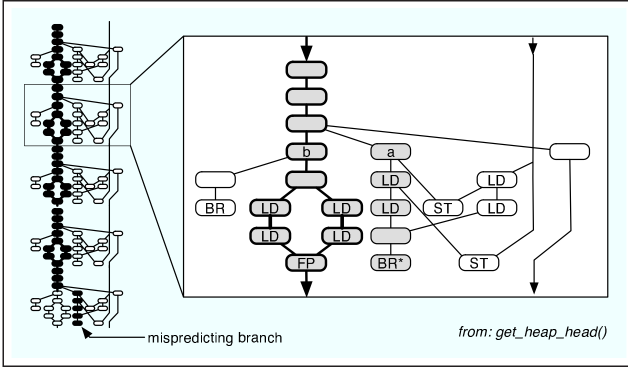


Figure 7. A code example demonstrating the source of contention-related stalls. The critical path, which ends in a mispredicted branch (**BR***), is highlighted. Both instruction **a** (on the rib) and **b** (on the spine) are predicted critical, but instructions on the spine are actually critical more often.

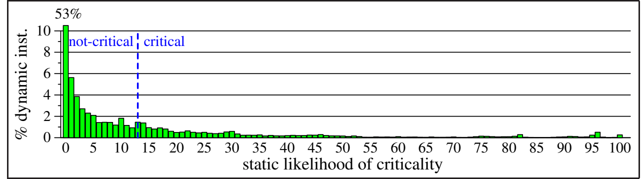


Figure 8. Distribution of LoC values. Data is averaged across all 12 benchmarks.

ity (LoC) metric. We assign an LoC of $n\%$ to a static instruction if $n\%$ of all previous dynamic instances of that instruction have been critical. That is, an instruction's LoC is the frequency at which it has been critical in the past. While this does not tell us the true criticality of any given instance of an instruction (that would require knowing exactly which dynamic branches are mispredicted, for example), we find that past criticality is a good indicator of future criticality. To verify that this is indeed the case, we modified our idealized list scheduler (discussed in Section 2.2) to prioritize instructions based solely on LoC values. In other words, we remove from the list scheduler its knowledge of *instantaneous* criticality, replacing it instead with knowledge only of average previous criticality. The results (not shown) indicate that the impact on the scheduler is marginal: the average performance loss moves from $\sim$1% to 1.5% and from 2% to 2.7% for the 4x2w and 8x1w configurations, respectively; the performance loss on the 2x4w configuration remains unchanged at less than 0.5%. In contrast, equipping the scheduler with only a binary notion of criticality shifts the performance losses to 1.5%, 5% and 9.8% on the 2-, 4- and 8-cluster configurations, respectively.

Figure 8 shows that the LoC metric yields quite a wide distribution of values, indicating that it has the potential to distinguish numerous degrees of criticality. The vertical dashed line in the figure shows the granularity achieved by the binary predictor used by Fields *et al*.[6] To understand the benefit of being able to distinguish various degrees of criticality, it is useful to view an LoC value as a measure of the expected cost of making a wrong decision on an instruction. For example, imposing a 2-cycle forwarding delay on an instruction with an LoC of 80% would, for each instance of that instruction, add about 1.6 cycles to the program's execution time. Likewise, an instruction with an LoC of 25% would incur 0.5 cycles for each instance. Being able to distinguish these two otherwise equally critical instructions, and hence prefer the former, potentially saves 1.1 cycles on each instance.

The utility of the LoC metric stems from the fact that it succinctly expresses dynamic behavior in terms of a single, numerical property of each static instruction. In contrast, a metric like slack [9] is much harder to express as a static property. This is because slack is measured as a cycle count for each dynamic instance, so different instances can have very different slack values. For example, branches, when mispredicted, have no slack; when predicted correctly their slack is very large, limited only by the size of the instruction window. From a static instruction point of view, this variation can be expressed as a histogram of the different amounts of slack observed by an instruction, but comparing two instructions on that basis is not very practical.

In Section 7, we show that the richer criticality information afforded by LoC can be used to reduce the number of contention-related stalls by a factor of two. Perhaps more importantly, LoC provides a general metric for controlling the resource allocation that we employ in Sections 5 and 6.

5 Stall over steer

In Section 3, we demonstrated that the dominant source of inter-cluster forwarding latency on the critical path is load-balance steering. In this section, we show how this problem can arise in regions of code that are execute-critical.

We begin with a hypothetical program that consists entirely of a single chain of dependent add instructions. Such a program has an ILP of 1 and no branch mispredictions, so it can be fetched much faster than it can be executed. In other words, the program is *execute critical*; the fetch rate is not affecting the execution rate. On a clustered machine, this will result in the dependence chain filling the window of the cluster to which it is steered. When this happens, the existing dependence-based steering policy reacts by load-balancing — it will assign the next instruction in the chain to the least-utilized cluster. And so the process repeats: upon filling the second cluster, load-balancing redirects the chain to a third cluster until it too fills. As Figure 9 shows, the net effect of this load-balancing is the introduction of one forwarding delay into the dependence chain every N instructions, where N is the size of a cluster's window.

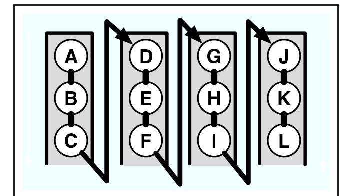


Figure 9. Load-balance steering causes a single dependence chain to be spread across all of the clusters. Performance would be better if steering had stalled when the desired cluster was full.

[6] The Fields predictor uses a 6-bit saturating counter that increments by 8 when training critical, and decrements by 1 when training non-critical; the threshold value for predicting critical is 8. Thus, 1 in 8 instances being critical is sufficient for an instruction to be classified as critical.

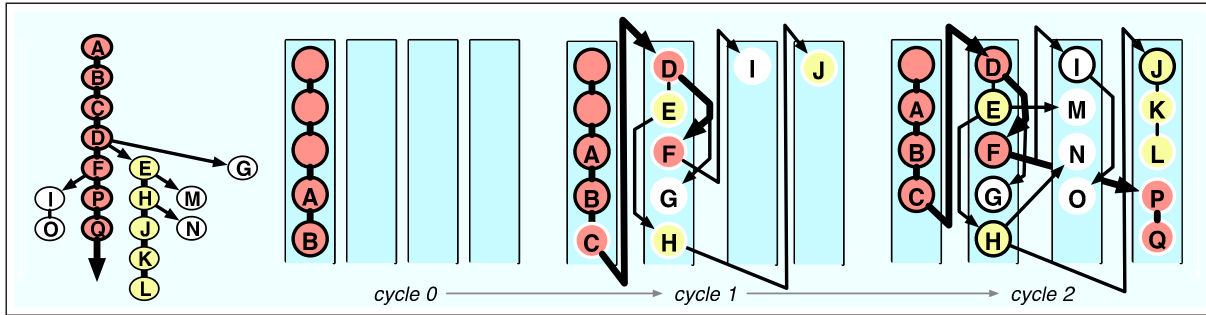


Figure 10. An illustrative example from `vpr` exhibits spreading of the critical path across clusters. The dataflow graph on the left shows the fetch order (alphabetical order) and the relative criticality of nodes (darker is more critical); instruction **L** is a frequently mispredicted branch. On cycle 0, the cluster containing the spine fills up. On cycle 1, one instruction of the spine executes, making room for **C**, which fills the cluster. **D** is steered to another cluster and is followed by **E**, **F**, **G**, and **H**. Because the second cluster is now full (for illustration, we are assuming 5 entries per cluster), **I** and **J** are assigned to different clusters. On cycle 2, instruction **P** cannot be steered to the second cluster, where **F** resides. In this example, the critical path **ABCDFPQ** incurs two unnecessary forwarding delays.

For this hypothetical piece of code, it would be preferable to *stall* the steering logic until a space becomes available at the desired cluster. Doing so would not slow the program down because, as we noted already, the fetch rate is not determining the execution rate. Instead, stalling would eliminate the forwarding delay from the dataflow chain entirely, permitting the clustered machine to perform equivalent to a monolithic one.

Real programs are of course not as simplistic, but this type of behavior does indeed occur. Figure 10 is an example of how the critical path in our `vpr` code sample (from Section 4, slightly simplified) incurs even more forwarding delays than the single dependence chain. We believe this is the effect that is responsible for the observation, made by Balasubramonian *et al*, that low ILP programs perform better on 4 1-wide clusters than on 16 1-wide clusters [2]. In effect, a smaller number of clusters increases the chance (from 1-in-16 to 1-in-4) that critical dependences are steered to the same cluster when the load-balancing occurs.

The potential benefits of stalling the front-end have also been reported by González *et al* [11]. They propose a scheme that uses the number of in-flight instructions at each cluster as a means for controlling the decision to stall. However, cluster load is a very coarse, and potentially misleading, measure of the phenomena that make stalling beneficial. In our hypothetical code example, stalling makes sense because the program is in an execute-critical region. In contrast, some program regions are *fetch critical*, in the sense that they are not critical themselves, but they need to be fetched as fast as possible to reach a critical computation, such as a mispredicting branch slice, that occurs in the future. For fetch-critical code, load-balancing is preferable when a desired cluster becomes full because it does not delay fetching of critical computations that lie ahead in the instruction stream.

To distinguish cases where stalling is preferable to steering, we can use the LoC metric. Instructions with a high LoC are likely to be execute critical, so stalling is preferable; those with a low LoC are probably fetch critical, in which case stalling would do more harm than good. Empirically, we find that stalling instructions with an LoC exceeding a 30% threshold strikes a good balance. Figure 11 shows how the `vpr` example behaves when nodes with high LoC values are stalled and all others are load-balanced.

A second benefit of the stall-over-steer policy is also made apparent in Figure 11: the machine is now doing a better job of distributing parallel work among the clusters. For example, instructions **E**, **G**, and **I** are not sufficiently critical for stalling, so they are load-balanced to other clusters where they can execute in parallel with the primary dependence chain. In contrast, the non-stalling scheme sends all of **D**'s successors to the same cluster, where they will contend for a single issue slot. This leads us to our next observation about load-balancing.

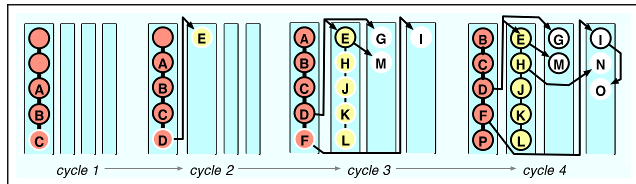


Figure 11. Stalling steering for execute-critical instructions prevents cross-cluster penalties. Starting from the cycle 0 state in Figure 10, in cycle 1 only instruction **C** is steered, because **D** is critical and its desired cluster is full. In cycle 2, there is space for **D** and, because **E** is not sufficiently critical, it is load-balanced to the second cluster. **F** is steered in cycle 3, after which non-critical **G** and **I** are load balanced, and **H**, **J**, **K**, and **L** are dependence-based steered to cluster 2. In cycle 3, instructions up to **P** are steered, but **Q** must wait until the next cycle. While this approach does not fetch the program as quickly, it preserves the critical slice on one cluster, and there is no benefit to fetching the program faster than it can execute.

6 Proactive Load-balancing

We have just seen that a positive side-effect of the stall-over-steer policy is improved load-balancing of less critical instructions. While this is the desired load-balancing behavior, relying only on the selective stalling mechanism to achieve it has a number of drawbacks. First, it only occurs after a cluster has filled — it is *reactive*. Second, it requires that there be a clearly identifiable critical dependence chain; some high-ILP regions have no instructions with sufficiently high LoC. Finally, the load-balancing applies only to non-critical instructions that diverge from the main dependence chain; instructions that diverge from other, less-critical chains do not likewise benefit.

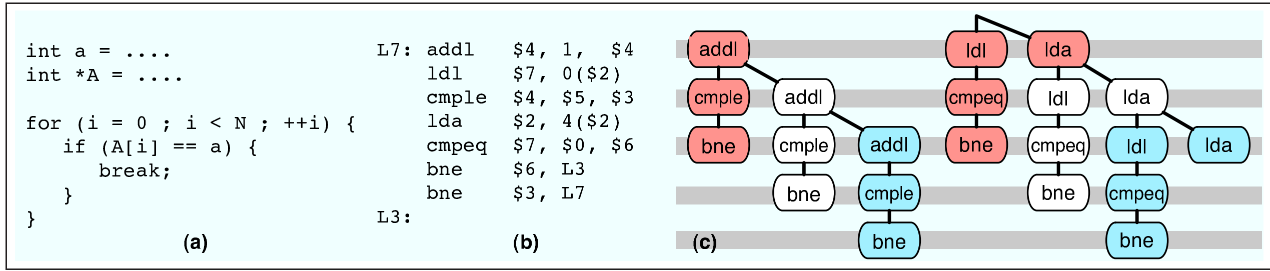


Figure 12. An example loop with divergent dataflow. **(a)** C-code for a loop with an early exit. **(b)** The corresponding Alpha assembly, which has been optimized by the compiler to have two separate loop-carried dependences. **(c)** A dynamic dataflow graph with three iterations of the loop (shaded in different colors) that shows how the instructions diverge from the loop-carried dependences.

The code example in Figure 12 demonstrates why a more general, *proactive* load-balancing scheme is needed. When dynamically unrolled, this code forms two trees of diverging dataflow. Because a dependence-based steering mechanism will try to collocate instructions with their dataflow producers, each tree will be assigned to a single cluster until it fills. This is clearly not the desired behavior. For example, on the 8x1w configuration, it will introduce contention stalls that will cause the branches to resolve 5 to 10 instructions later than they would have on a monolithic machine.

One approach that has been adopted for proactive load-balancing is steering only the first dependent instruction to a given producer; all others are load-balanced [15, 19]. While this technique does a good job of distributing the consumers of an instruction, it has the drawback that it can introduce forwarding delay onto the critical path. This will happen when the first consumer — the one that gets collocated with the producer — is not the most critical one. For example, in the code sample of Figure 12, the most critical consumer of the loop-carried dependence produced by the `addl` instruction is the next instance of itself, yet this is also the last consumer of that value (it must be because it performs a destructive update of the register). If the forwarding delay is 2 cycles, steering this critical consumer to a different cluster reduces the IPC of this code from a potential peak of 7 down to 7/3, as shown in Figure 13(a). Contrived though this example might at first seem, we find that the potential for this behavior is exhibited broadly across the SPEC integer benchmarks: of all the critical instructions that have multiple consumers, more than 50% do not have their most critical consumer first in fetch order.

Figure 13(b) shows a more preferable distribution of instructions. In this case, the most critical consumer is retained at the producer's cluster and the less critical consumers are load-balanced. The steady-state ILP is once again 7, and branch resolutions are delayed only two cycles, which is as good as can be done on a machine with single-issue clusters.

Achieving this schedule is challenging on a machine that performs steering in fetch order — it requires knowing, in advance, which consumer is the critical one and, by implication, which consumers are not. Since it is not our objective in this paper to discuss mechanisms, we will not attempt to propose a scheme for identifying the most critical consumer. The results we present in Section 7, which demonstrate the potential effectiveness of a proactive load balancing *policy*, are therefore based on a scheme that is unlikely to be practical in hardware.

That said, we do believe that a practical scheme can be developed. We are led to that conclusion by an analysis of producer-consumer relationships in our simulator's execution traces. Two properties of dataflow — one viewed from the producer's perspective and the other from that of the consumer — were uncovered by that analysis.[7] First, the most critical consumer for a given producer tends to be statically unique. We found that about 80% of all values produced can be associated with a statically unique most-critical consumer. Second, a given consumer tends either to always be the most critical consumer of its producer's value, or is almost never the most critical one. That is, there is a bimodal distribution of static consumers' tendency to be the most critical one. While this analysis is not conclusive, it does bode well for dynamic schemes that either associate producers with their most-critical consumer, or that tag consumers as the most critical for their operand.

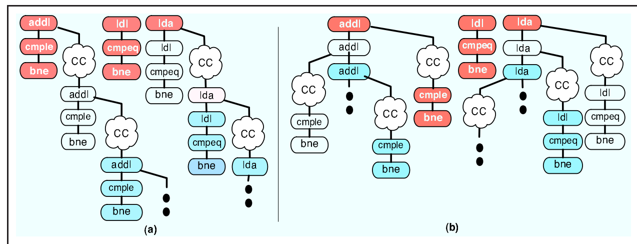


Figure 13. Load-balancing subsequent consumers can seriously impact ILP. Using the branch computation from Figure 12: **(a)** if only the first consumer is collocated, the recurrence will be spread across clusters, incurring the cross-cluster (**CC**) forwarding penalty; **(b)** in this case, the first consumer should be load-balanced to retain the "spine" on a single cluster to preserve the steady-state ILP.

7 Performance Results

In previous sections, we diagnosed and proposed policies for remedying the IPC penalties incurred by clustering. Our objectives in this section are twofold. First, we aim to give empirical evidence in support of our assertions that the previously identified causes for performance loss are, in large part, responsible for the IPC difference between existing policies and what is potentially achievable (as described in Section 2.2). Second, we aim to demonstrate that the policies we have defined are indeed effective at mitigating those

[7] Due to space constraints, we omit plots of our results and instead summarize the main trends we observed.

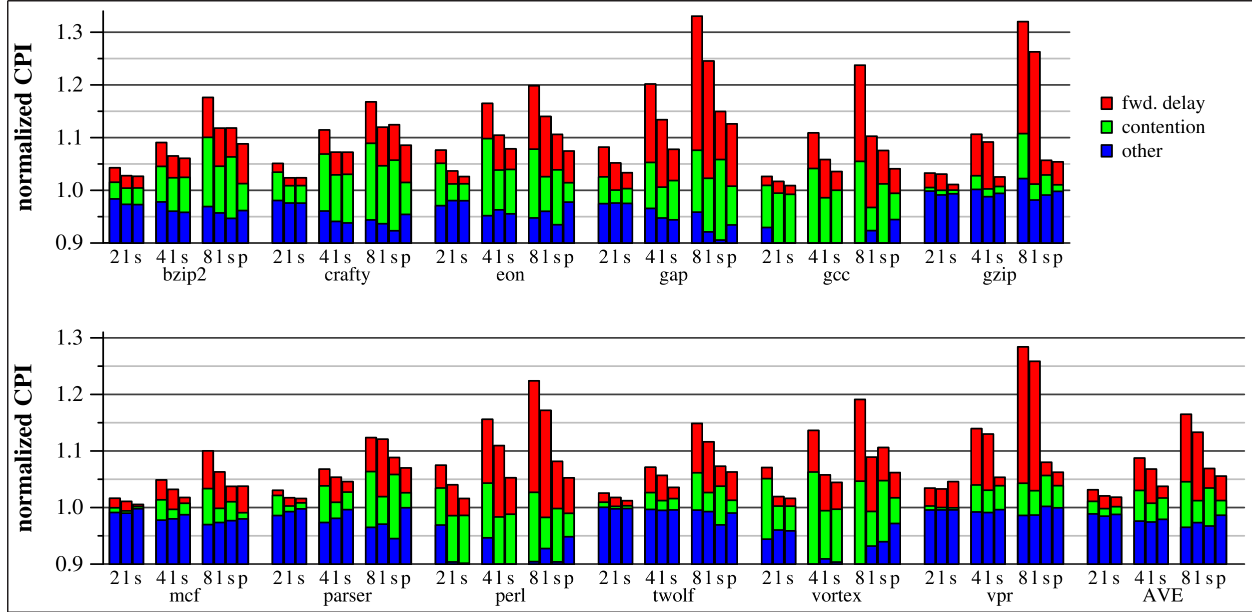


Figure 14. The proposed policies reduce the penalty of clustering by 42%, 57%, and 66% for 2-, 4-, and 8-cluster machines, respectively. The first bar for each configuration (**2**, **4**, and **8** clusters) reports the performance of Fields *et al*'s focused steering and scheduling (also shown in Figure 4). The bars labeled **l** add LoC-based scheduling. The bars labeled **s** add stalling rather than load-balancing for high LoC (execute-critical) instructions. The bars labeled **p** add proactive load balancing to the 8-cluster case (our implementation does not benefit the wider clusters). All bars are normalized to a monolithic (1-cluster) machine that uses LoC-based scheduling.

performance problems. To this end, we have implemented our policies in the timing simulator and evaluated their performance on the Spec Integer benchmarks. We stress that *we are not advocating these particular implementations*; our goals, as noted already, are merely to substantiate our claims quantitatively.

We implemented a likelihood of criticality predictor by tracking the fraction of executions that were detected as critical, using the critical path profiling framework developed by Fields *et al* [10]. In other work, we have found that stratifying LoC into 16 levels produces results almost equivalent to a counter with unlimited precision. Intuitively this makes sense, as 16 levels generally distinguishes two instructions that differ in criticality by as little 7%; this will only fail to select the more critical instruction when two instructions have very similar likelihood of criticalities. Using probabilistic counter updates [21], we have implemented a predictor that stratifies these 16 levels using just 4 bits of storage, less space than the 6-bit counters used by Fields *et al*.

With the LoC predictor in place, implementing criticality-based selective stalling is quite straightforward. As previously noted, we use a 30% LoC threshold to choose stalling over load-balancing.

The proactive load-balancing policy is the most challenging to implement in a fully dynamic steering mechanism. Briefly, our implementation works as follows. To decide which consumers should be load-balanced, we track the most critical consumer of each register at steering time. When a consumer retires, we compare its LoC with that of the most critical consumer thus recorded; if the value is lower, we tag the consumer as a candidate for load-balancing. On top of this, we generally steer only one consumer to the same cluster as a given producer. This is accomplished by tagging the producer when it has been followed, so that it will be ignored by future consumers. We override this single consumer policy when a particularly critical consumer is encountered — we refuse to load-balance an instruction if its LoC is greater than 5% and it is at least half as critical as the producer (suggesting that it is the most critical consumer). While we do not believe this constitutes a reasonable implementation, it is one that does not require any oracular knowledge, suggesting that a realistic implementation exists.

Figure 14 demonstrates the benefit of instituting these policies in our simulated clustered machines. The LoC-based scheduling is clearly advantageous — for all benchmarks and all configurations it provides a speedup relative to scheduling based on binary criticality. On average, it halves the execution time lost to contention-related stalls and, in some benchmarks (*e.g.*, `gap`, `gcc`, `vortex`), it indirectly reduces the number of forwarding stalls substantially.

In contrast, the stall-over-steer policy is not universally beneficial, but it provides a substantial benefit to `gap`, `gzip`, `perl`, and `vpr`, which are some of the programs that previously saw the biggest penalties from clustering. Much of the 20% speedup this policy achieves in `gzip` on the 8-cluster machine occurs in long stretches of the execution where only 3 clusters are used. This confirms our earlier observation that cluster utilization is not a metric to be optimized. The negative impact on `crafty` and `vortex` is small, but it suggests that there is potential to further tune this policy. In general, stall-over-steer leads to a substantial reduction in the critical forwarding latency, but it occasionally aggravates the number of contention-related stalls. This occurs when non-critical instructions that are assigned to the "critical cluster" are delayed too long, a problem that is mitigated by proactive load-balancing.

As implemented, our proactive load-balancing policy only benefits our 8-cluster machine; the other configurations have multiple-issue capabilities at each cluster, so they are less sensitive to load imbalance. This policy improves performance by eliminating con-

tention stalls from near critical paths, reducing both the number of critical contention stalls and the frequency at which near-critical paths are made critical. However, it offers little benefit to programs like `gzip`, `twolf` and `vpr`. We believe this is related to the fact that both of these programs have dataflow hammocks on the critical path, as seen in Figure 7. In such cases, it is often better to execute on a single cluster because of the latency of inter-cluster communication.

Overall, our three policies are sufficient to bring all cluster configurations to within 5% of the performance attained by the list scheduler used in the idealized study of Section 2.2. The obvious question then arises as to what prevents us from removing the remaining 5%. Since we did not attempt to explore the design spaces opened up by each of our policies, there is almost certainly room for improvement in the performance we get from them. However, this is not likely to account for all of the remaining performance loss. We believe that the bulk of the 5% loss results from an inefficient distribution of ready instructions across the clusters. Although proactive load-balancing pushes subsequent consumers to different clusters, to achieve optimal load balance, these instructions must be assigned to a cluster that does not already have (and will not soon have) ready instructions. In other words, choosing the least-full cluster in these circumstances is not always appropriate.

The problem is most challenging when the program code exhibits ILP equal to the machine's width. Figure 15 shows how effective our policies are at extracting all of the available ILP on the 8x1w configuration. When the available ILP is exactly 8, a clustered machine needs to have distributed one ready instruction to each of its clusters. This is particularly challenging because, in the presence of non-unit latency instructions, it implies the need to assign more than one dataflow chain to a cluster, each with ready instructions on interleaving cycles. Achieving such a fine-tuned load balance seems to require tracking exactly when and where each instruction will be ready. This is exactly what our list scheduler does — it makes its cluster assignment decisions based on a global view of all in-flight instructions. In the clustered machines we study, steering is decoupled from scheduling, so cluster assignments must be made in the absence of precise information.

However, the problem becomes much easier when available ILP is either very high or very low. For example, with an available ILP of 24, we are likely to have 3 ready instructions per cluster on average, making it unlikely that there will be cycles when we do not achieve full utilization of the machine. When available ILP is low, there are few dataflow chains, so each can be assigned to its own cluster, guaranteeing that all ready instructions will be executed every cycle.

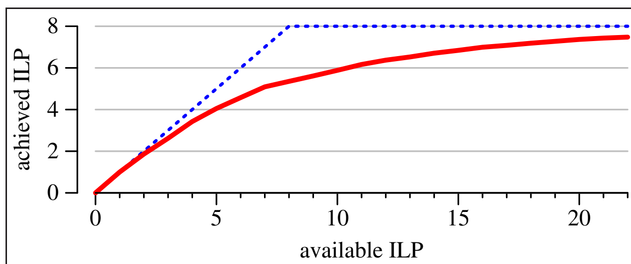


Figure 15. A clustered machine has trouble matching a monolithic machine when code has ILP close to the total issue width. Data shown is an average over all SPEC Integer benchmarks for the 8x1w machine. *Available ILP* is computed on a cycle-by-cycle basis by counting the number of ready instructions across all clusters. The *achieved ILP* is the average number of instructions executed on cycles with a given available ILP.

8 Conclusion

Clustered machines are an attractive design option because they address the power-, clock- and complexity-related problems that hinder scaling of monolithic designs. However, all studies to date show a significant IPC penalty incurred by clustered designs. Motivated by the question as to whether those penalties are inherent or merely a result of sub-optimal use of the hardware, we conducted an idealized study in which we list-schedule instruction traces for various clustered machine configurations. In terms of inherent performance capabilities, our findings are positive: for a family of clustered architectures and a variety of inter-cluster forwarding latencies, there exist instruction schedules that yield performance remarkably close to that of an equivalent monolithic machine. This result indicates that, in the integer applications we examined, critical dataflow exhibits very modest ILP; equally, non-critical dataflow is very tolerant of inter-cluster communication penalties and intra-cluster resource contention. In short, if critical dataflow can be accurately identified and appropriately co-located, then a clustered architecture imposes negligible IPC penalty.

With this result in place, we used critical-path analysis to examine a state of the art policy for steering and scheduling, aiming to discover the main causes that underly the observed performance penalties. We found two main culprits: contention among known-critical instructions and load-balancing of critical instructions. These findings motivated the development of policies for mitigating the performance-degrading effects. Regarding contention among critical instructions, we found that a binary notion of criticality is not sufficient for correctly prioritizing among critical instructions. We introduced the likelihood of criticality (LoC) metric to address this problem, and showed that a scheduler equipped with LoC information can, on average, halve the time lost to contention stalls. The load-balancing problem poses a more difficult challenge. When programs are in execute-critical regions that are clearly execute-critical, we showed that a stall-over-steer policy can eliminate forwarding delay from the critical dependence chain. Using the LoC metric as a means for driving the decision to stall rather than steer, we showed that performance in some benchmarks can be improved by as much as 20%. We also observed that a stall-over-steer policy effects a better distribution of ready instructions, but found that its potential in this respect is limited by fetch order constraints and codes in which a clearly-critical dependence chain is not discernible. To tackle that problem, we proposed a proactive load-balancing scheme that pushes non-critical consumers away from their producers, leaving room for the more critical consumers.

Together, these policies bring performance of all of the clustered configurations we studied to within 5% of our idealized list scheduling. We believe the remaining performance gap results from the fact that scheduling in a dynamic clustered machine is distributed and decoupled from steering. In the absence of a global and accurate view of instruction readiness, it is difficult for steering to optimally distribute the ready instructions among the clusters. This problem is most pronounced on a machine with 1-wide clusters when the

code it executes has ILP close to its aggregate execute bandwidth. From an IPC point of view, therefore, a machine with 1-wide clusters is less appealing than the wider configurations we examined. Since 2-wide clusters are only moderately more complex, they are perhaps the more complexity-effective design point.

We believe the policies we have developed in this work move us one step closer to an effective dynamically-scheduled clustered machine. Nevertheless, a number of serious implementation challenges remain. For example, dynamic profiling of the critical path requires that a token-passing predictor be built into the pipeline; implementing the LoC scheduling policy potentially complicates the dynamic scheduler; and dynamically identifying which instructions should be proactively load-balanced will not be easy. In fact, even building a circuit that can do dependence-based steering of 8 instructions per cycle is not likely to be easy — it suffers the same complexity-related problems incurred by register renaming logic (namely, intra-cycle dependences need to be taken into account). Given these hurdles, we are currently investigating hardware-software hybrid techniques that might enable a feasible implementation.

Acknowledgments

This research was supported in part by NSF CCF-0429561, NSF CCR-0311340, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. We thank Sarita Adve, Vikram Adve, Krishnan Kailas, Andrew Lenharth, Sanjay Patel, and the anonymous reviewers for feedback on this work.

References

[1] A. Aletà, J. M. Codina, A. González, and D. Kaeli. Instruction Replication for Clustered Microarchitectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[2] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[3] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 337–347, Dec. 2000.

[4] R. Bhargava and L. K. John. Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[5] R. Canal, J. M. Parcerisa, and A. González. Dynamic Cluster Assignment Mechanisms. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Department of Computer Science, Yale University, 1985.

[7] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1997.

[8] B. Fields, R. Bodik, M. Hill, and C. J. Newburn. Using Interaction Cost for Microarchitectural Bottleneck Analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[9] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing Performance Under Technological Constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[10] B. A. 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.

[11] J. González, F. Latorre, and A. González. Cache Organizations for Clustered Microarchitectures. In *WMPI '04: Proc. 3rd Workshop on Memory Performance Issues*, June 2004.

[12] K. Kailas, A. Agrawala, and K. Ebcioglu. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[13] G. A. Kemp and M. Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proceedings of the International Conference on Parallel Processing*, pages 239–246, Aug. 1996.

[14] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[15] H.-S. Kim and J. E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *ISCA*, Alaska, May 2002.

[16] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O. Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[17] S. Narayanasamy, H. Wang, P. Wang, J. Shen, and B. Calder. A Dependency Chain Clustered Microarchitecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[18] E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *31th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-31)*, Nov 1998.

[19] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[20] S. Palacharla and J. Smith. Decoupling Integer Execution in Superscalar Processors. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1995.

[21] N. Riley and C. Zilles. Probabilistic Counter Updates for Predictor Hysteresis and Bias. *Computer Architecture Letters*, August 2005.

[22] J. E. Smith. Decoupled Access/Execute Computer Architecture. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 112–119, Apr. 1982.

[23] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.