CHAPTER 7

A CLUSTERED FRONT-END FOR SUPERSCALAR PROCESSORS

This chapter studies techniques for distributing the main components of the processor front-end with the goals of reducing their complexity and avoiding replication, so they extend the advantages of clustering to structures like the branch prediction, instruction fetch, steering and renaming. In particular, effective techniques are proposed to cluster the branch predictor and the steering logic, which minimize the wire delay penalties caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation, and the cluster assignment. By reducing the latency of these critical loops, clustering results in faster clock rates, or bigger structures with the same clock rate. The schemes proposed in this paper to deal with inter-cluster communications are very effective since they reduce communication penalties to just a 4% IPC degradation, for SpecInt95.

7.1 Introduction

Clustering of computational elements [42, 70, 99, 105] is an effective method for dealing with scaling, complexity [70], power [112], heat, and clock distribution [47] problems. Previous clustered microarchitecture proposals [18, 29, 34, 52, 53, 54, 70, 73, 82, 112] have focused on clusters containing register files, functional units, and issue queues. However, in that previous work, the processor front-end units (e.g. instruction fetch, decode, rename) are centralized in a conventional manner. However, the advantages of clustering apply to any processor component. In the research reported here, we propose and study the design of front-ends containing clustered subsystems. The branch predictor, instruction cache, decode and rename logic are all either distributed, or in some cases replicated, and grouped into a number of clusters.

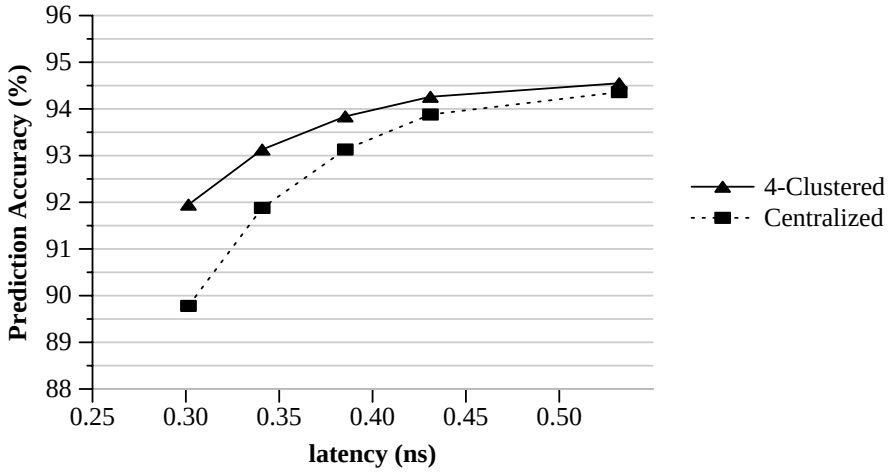

Figure 7-1: Accuracy vs. latency for a centralized and a 4-clustered predictors.

Note that the advantages of clustering apply to any processor component. For instance, let us consider the branch predictor. Because it is involved in the task of the fetch address generation, which forms a critical hardware loop, the predictor size -hence its accuracy- is greatly limited by cycle time. By partitioning the predictor into four smaller predictors, the latency of each one is considerably lower. Conversely, the effective size of a clustered branch predictor can be made four times bigger without increasing its latency. Figure 7-1 illustrates this by comparing the effectiveness of a clustered branch predictor to that of a centralized one, for different predictor latencies (sizes)[1]. As it can be seen, clustering results in important accuracy improvements for a given latency, or significant latency reductions for a given accuracy. There are many techniques, such as banking, that may optimize the access time of a predictor table, but they are orthogonal to our approach, because they equally apply to both a centralized predictor and each partition. Our approach to partitioning the branch predictor goes one step further because it leaves cross-structure wire delays out of the critical path of the fetch address generation loop by converting them to cross-cluster communications.

As a second example, let us consider the steering logic in a superscalar architecture with a clustered back-end. This piece of logic is located in the front-end and takes care of selecting the most appropriate execution cluster for each instruction, prior to steering it to the issue queues. A cluster assignment algorithm that steers instructions according to data dependences is key for performance, especially if the register file is also distributed. Unfortunately, this kind of algorithm serializes the steering of a sequence of instructions because the assignment of each instruction depends on previous assignments. For an eight-way or wider superscalar, this task may possibly take more than a single cycle. However, because the assignments of one fetched block of instructions are needed to steer the next block, the steering task cannot be spread into several stages without generating pipeline bubbles. Our approach is to partition the assignment

1. This experiment assumes a gshare+bimodal hybrid predictor with equally sized tables. Table sizes range between 0.25K and 64K entries. The latencies are calculated for a 90nm tech. and for layouts optimized for speed, using a modified version of the CACTI 3.1 tool (http://research.compaq.com/wrl/people/jouppi/CACTI.html).

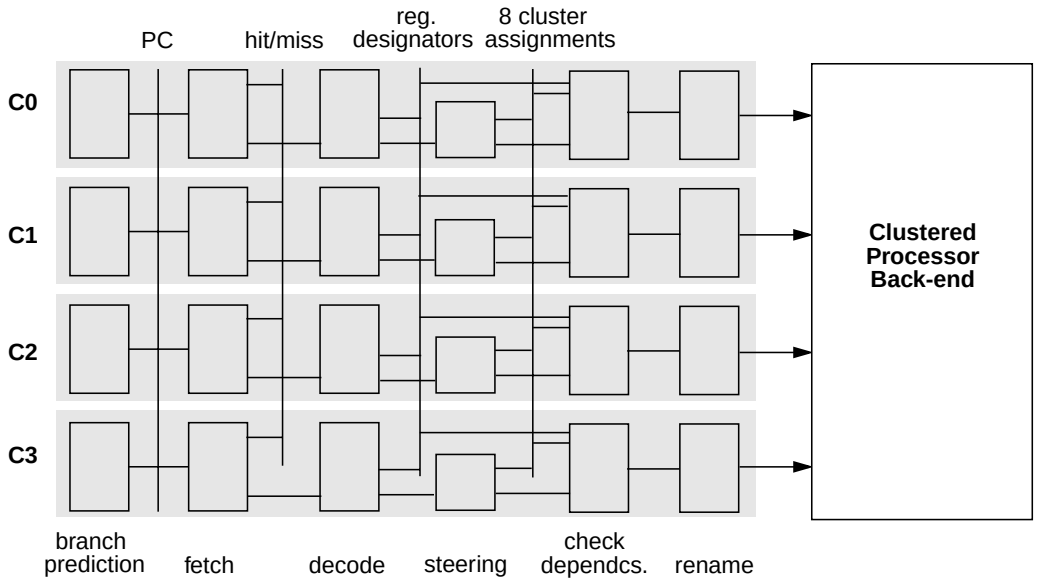

Figure 7-2: Partitioning an 8-way processor's front-end into four clusters.

task into smaller units that operate in parallel, thus reducing the amount of work of each partition and shortening the total latency of the cluster assignment logic.

To summarize, in this work we explore techniques for clustering each of the major front-end components with the objective of minimizing replication and inter-cluster communication. Even if a designer chooses not to implement a fully clustered front-end, this research points to new ways of partitioning the individual front-end components, such as the branch predictor, that could be used in an otherwise conventional front-end.

We evaluate the performance impact of clustering front-end structures by comparing with a baseline processor composed of a conventional centralized front-end and a clustered back-end (similar to the one described in chapter 2). To account for longer inter-cluster delays, we follow the strict rule of adding a full clock cycle to any paths that involve inter-cluster communication. Ignoring the clock cycle advantage, the proposed clustered front-end has about the same performance as the non-clustered one. For instance, we observe that the average IPC of the clustered organization for SpecInt95 is within 4% of the non-clustered one.

7.2 Clustering Front-End Subsystems

As a vehicle for developing and studying clustered front-end microarchitectures, we begin with a conventional eight-way superscalar microarchitecture consisting of a clustered back-end and a front-end divided into four clusters, each one capable of processing two instructions per cycle (see Figure 7-2). Consequently, all our discussion will focus on this particular 4-by-2 configuration but the mechanisms can be applied to any other (m-by-n) configuration.

Because the I-cache and branch predictor tables are *partitioned* among the clusters, each front-end cluster holds its own PC, a portion of the I-cache and a portion of the branch predictor. The rename table, in contrast, is *replicated* in all clusters, but each copy has fewer read ports than a centralized implementation. Because the back-end is clustered, the front-end also performs a steering function aimed at placing dependent instructions within the same back-end cluster, and the cluster assignment logic is also *partitioned* among the clusters. The pipelines implemented in the four front-end clusters work closely in parallel so that instruction blocks fetched during the same cycle advance through the clusters together. As noted earlier, we always assume a single-cycle latency for all signals that pass among clusters. The following subsections describe the approach taken in each of the stages of the clustered front-end pipeline.

7.2.1 Clustering the Branch Predictor (Stage 1)

As noted before, the branch predictor is part of one of the critical hardware loops [13] of any superscalar processor: the fetch address generation. The fetch address must be generated in a single clock cycle to avoid pipeline bubbles. Thus, the predictor size - hence its accuracy - is greatly limited by cycle time. Clustering the branch predictor may enable larger predictors without increasing their latency, or conversely it may enable faster clocks without losing accuracy.

Branch prediction in a clustered front-end introduces new challenges. As with many clustered units, there are two choices for implementing the branch predictor: replicate resources and have each cluster make the same prediction in parallel, or distribute resources and communicate results from one of the clusters to the others. The advantage of replication is that the communication latency between clusters is avoided. The disadvantage is that overall the predictor consumes four times the area with no additional prediction accuracy. The advantage of partitioning and distributing the branch predictor is that it is effectively four times bigger, but prediction results must be communicated to all the clusters from the one that is making any given prediction.

We chose to develop a partitioned design and to integrate it into the pipeline in such a way that the communication delay leads to negligible performance degradation. Our approach, which applies to both the BTB and branch direction predictors, places the predictors at the beginning of the pipeline and divides the predictors into four banks, one per cluster, interleaved by certain bits of the program counter. The program counter is replicated in all the clusters, so the time a predictor bank of size S takes to update its local copy is exactly the same as a centralized predictor of the same size S, i.e., a centralized predictor and a clustered one that is four times bigger fit into the same cycle time.

Each cycle, the prediction bank in one of the clusters is active, depending on the bank selection bits taken from the current fetch address. The active predictor is accessed and its prediction must be broadcast to the other clusters so that the replicated PCs are kept consistent. Because this communication is assumed to take a full clock cycle, the rest of PCs are updated one cycle later. While the active cluster does not change, the predictor may keep producing one prediction per cycle. However, when the predictor changes from one cluster to another, a one

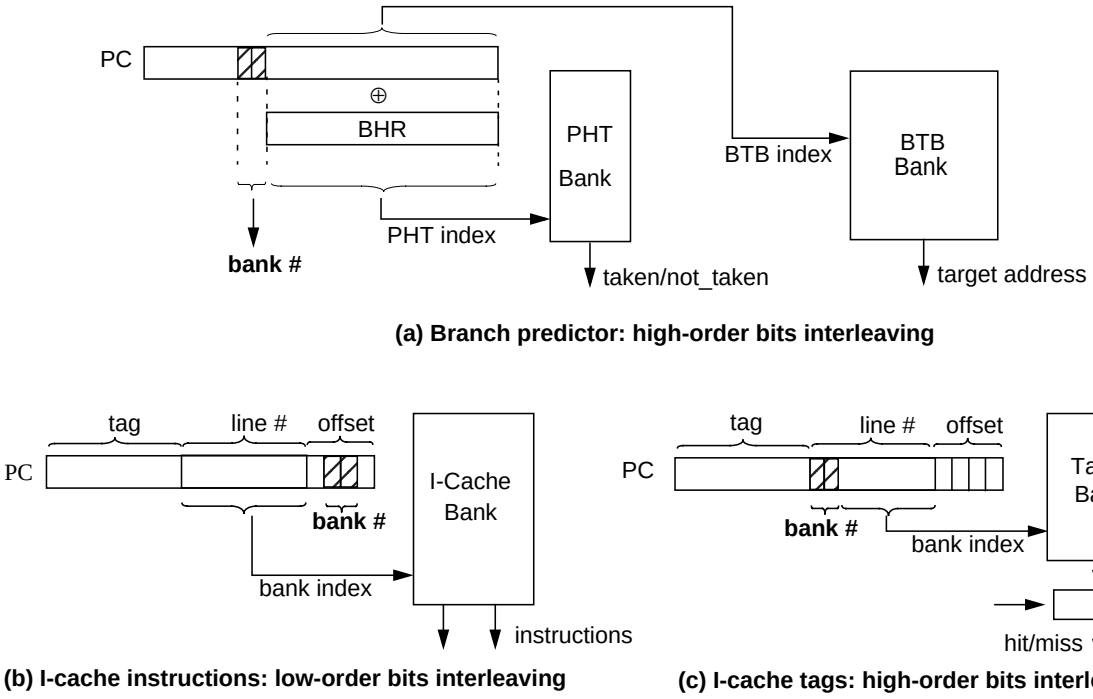

Figure 7-3: Branch predictor (gshare) and instruction cache interleaving

cycle bubble is inserted into the pipeline, because the next prediction cannot start until it can determine the next fetch address. Therefore, the BTB and the predictor tables must be interleaved using some *high order* bits of the address (figure 7-3a) because then, if the code has good spatial locality, it is likely that the same predictor bank is active for many consecutive cycles without creating many bubbles.

Bank switch occurrences could be further reduced by interleaving on even higher order bits of the address, beyond the bits of the index. However, if too high order bits are chosen, a particular code may then run entirely on a single or few predictors, possibly increasing conflicts and losing accuracy. Therefore, the choice of interleaving bits must trade cluster switch frequency for prediction accuracy, which is studied in section 7.3.2. However, for most of our experiments we just assume the scheme outlined in figure 7-3a.

While using the same address to build their index, the predictor and the instruction fetch cannot start at the same time, as it occurs in conventional architectures (figure 7-4a), because it takes one cycle to broadcast the PC to all clusters. In our clustered design the actual instruction fetch stage is delayed by one cycle (stage 2) so that the communication (shown as Bcast in figure 7-4b) may be pipelined smoothly. Note that the prediction stage only depends on the *local_PC*, not the *globally broadcast PC*, unless there is a bank predictor switch, then it depends on the broadcast PC, and a bubble is inserted in the pipeline (see figure 7-4c). Figure 7-5 depicts the pipelines of the baseline (centralized) and clustered front-end architectures.

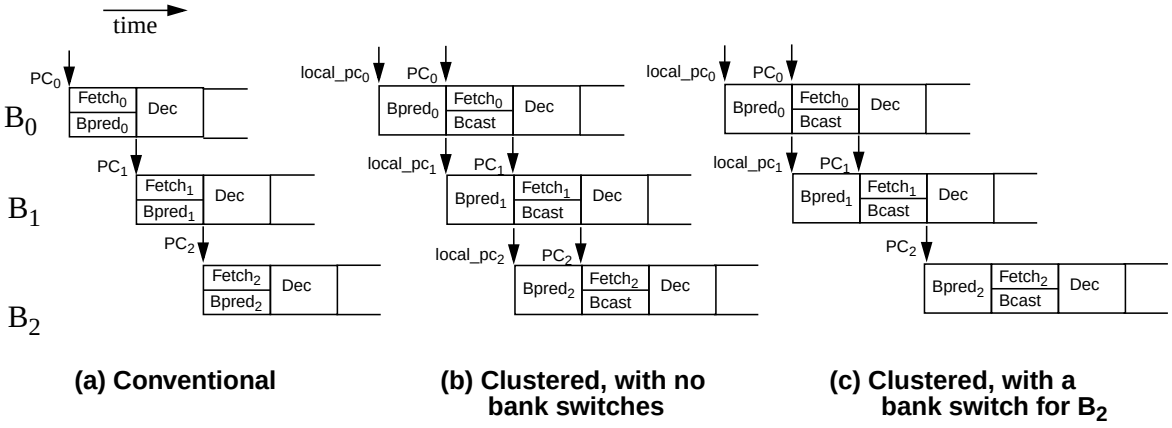

Figure 7-4: Pipeline timing of conventional and clustered front-ends. The diagram shows the fetch of 3 blocks of instructions B0, B1 and B2, at addresses PC_0, PC_1 and PC_2 respectively

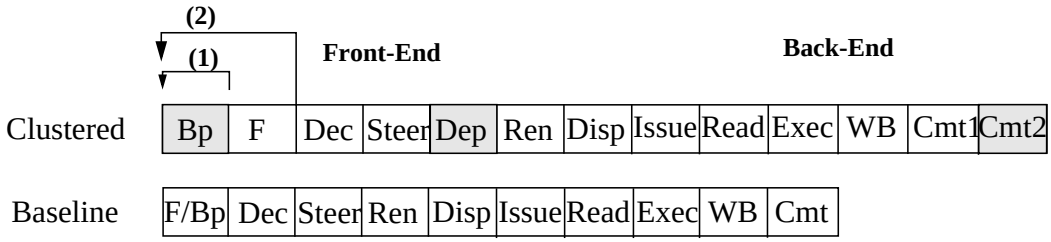

Figure 7-5: Clustered and Baseline (centralized) integer pipelines. Fetch address generation loop is (1) in both pipelines, but it becomes (2) in the clustered one when switching predictor cluster, producing a bubble.

7.2.2 Clustering the I-Cache (stage 2)

Assuming that the predictor is clustered as described above, clustering the I-cache is straightforward and becomes the natural choice for reducing structure size and wire length. Our proposed I-cache is partitioned into banks, which are distributed among the clusters. Each bank is accessed with the local copy of the PC. Hence, assuming equally optimized cache layouts, the fetch time of the clustered cache is exactly the same as that of a centralized cache with one fourth its size. To associate one decoder to each I-cache bank, cache banks must be block-interleaved, i.e. interleaved by some low order address bits of the line offset (see figure 7-3b), so that each bank delivers a portion of the cache line that includes one or more adjacent instructions (2 instructions, on our 8-way 4-cluster example).

The I-cache tag arrays are also clustered to get one tag bank next to each PC. No matter whether the tags are interleaved by the high order or the low order bits of the cache line index, only one bank is activated each cycle. However, for those designs that allow fetching instruction blocks spanning two consecutive lines, interleaving by the high order bits will be a better choice (see figure 7-3c), because it will increase the likelihood that consecutive lines map to the same tag bank. The active tag cluster generates a hit/miss signal and forwards it to the other clusters.

Due to the broadcast delay, this signal is not available to the other clusters until one cycle later. Hence, in case of a cache miss the instructions fetched during the current and the following cycles must be squashed.

An alternative design could interleave the branch predictor and cache tags with the same address bits. Then the active predictor and tag bank would be always in the same cluster, and the tag check could be anticipated during the prediction (stage 1), so that the hit/miss signal could be delivered to all clusters by the end of the fetch stage. This approach would be especially appropriate for set associative caches, where the way selection bits are sent along with the hit/miss signal.

7.2.3 Decode (stage 3)

Each cluster decodes 2 instructions per cycle in stage 3. In addition, the I-cache hit/miss signal is broadcast during this stage. To check dependences for renaming, the destination register designators of all eight possible instructions are broadcast to the other clusters, which takes one clock cycle, during stage 4.

7.2.4 Clustering the Steering Logic (stage 4) and Broadcast of Cluster Assignments (stage 5)

The steering logic assigns each instruction to one back-end cluster, where it will be later steered during dispatch. Note that in our assumed architecture, this task must take place before renaming. The rename logic allocates a free physical register for each instruction that produces a result. However, if the register file is distributed, there is a separate free list for each back-end cluster, and the rename logic allocates one physical register only from the free list of the cluster assigned to the instruction, hence this cluster must be known before renaming.

As noted before, dependence-based steering schemes minimize inter-cluster communications and has proven the most effective scheme for architectures with distributed structures in the back-end. However, it involves a serial task because the steering decisions for one instruction depend on the assignments made for previous instructions. Hence, fitting the delay of the steering logic into a single clock cycle may be difficult, but spreading it into more than one cycle may introduce bubbles into the pipeline, because the outcome for a given block of instructions is needed by the next one. The problem of reducing the total latency of the steering logic may be handled by parallelizing the steering task and partitioning it into clusters, in the following way.

Let us first consider a naive approach that partitions the steering logic into four clusters, and lets each cluster make the steering decisions for its two local instructions, independently of the assignments in other clusters, during stage 4. Each cluster performs just one fourth of the total task, with a great reduction of the total latency. The cluster assignments produced in each cluster are broadcast to the other clusters during the next cycle (stage 5), so they are available to the steering logic of all clusters after two cycles. The renaming, which needs to know the assignments of all eight instructions, may take place in stage 6, and the actual steering of

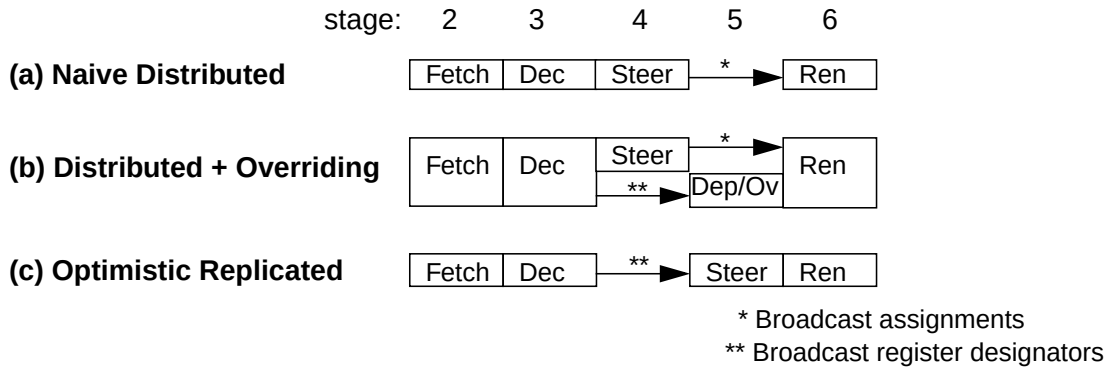

Figure 7-6: Pipeline diagrams for three clustered steering logic schemes

instructions to clusters and insertion into the issue queues occurs in stage 7 (figure 7-6a). The main drawback of this scheme is that the local steering logic in one cluster not only ignores the assignments made by other clusters for instructions of the same block but it also ignores the assignments made during the previous cycle because they are still being broadcast. In other words, the steering logic uses two-cycle outdated assignment information. As a consequence, some instructions get steered to clusters different to the producers of their source operands. Program correctness is still guaranteed because the required copy instructions are generated during the rename stage according to correct assignment information, which is already known at that time. However, we found that using outdated information to make steering decisions has a significant impact on performance (see section 7.3.3) because the distance between producers and consumers is usually short, which causes many extra communications.

Our approach makes a similar partitioning of the steering logic, using partially outdated assignment information but, unlike the naive scheme, the produced cluster assignments are considered provisional. Then, a register dependence analysis is performed in parallel, to check if an assigned instruction depends on one of the instructions whose assignments were ignored by the steering logic. If the producer of a value is found among the previous instructions then, when all assignments are known at the end of stage 5, the assignment of the producer overrides the provisional assignment (figure 7-6b), so that the instruction is steered to the same cluster as the producer.

In more detail, each cluster must check all possible dependences between its two local instructions and the preceding instructions belonging to the same or the previous fetched block (see figure 7-7). The dependence analysis compares the source and destination register names of these instructions. These names are first produced by the decoders in various clusters in stage 3, then they are broadcast to all clusters during stage 4, and the dependence analysis takes place during stage 5, in parallel with the broadcast of the assignments. At the end of stage 5, for each instruction, a multiplexer determines its final assignment by choosing among its provisional assignment or the assignment of some precedent instruction, depending on the results of the dependence check logic.

Finally, for comparison purposes, we also consider an idealized approach to clustering the steering logic that uses perfectly updated assignment information and thus it avoids

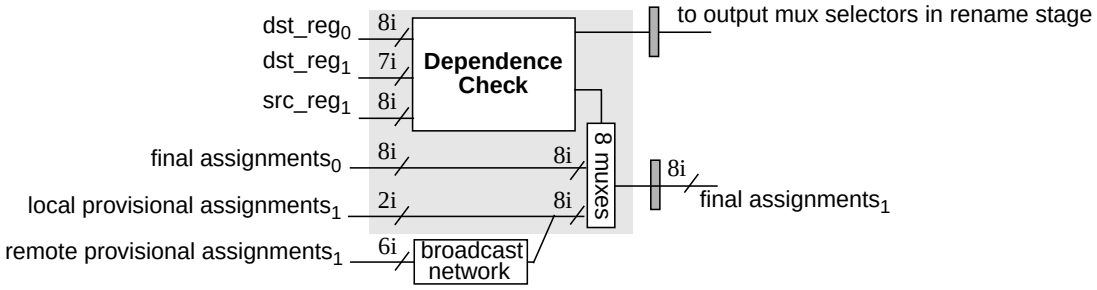

Figure 7-7: Block diagram of the Dependence Check and Overriding (stage 5) for one cluster. Signals with subscript 1 belong to the current block of 8 instructions. Signals with subscript 0 belong to the previous block

performance degradation. This scheme replicates the steering logic in all clusters, so that each cluster makes identical cluster assignments for all eight instructions. Since the required source register designators are broadcast in stage 4, the cluster assignments take place in stage 5 (figure 7-6c). Note that this scheme is optimistic because it assumes that steering eight instructions can be done in a single cycle.

As described before, the steering logic uses DCOUNT workload counters to measure workload balance. These counters are updated during the steering stage (stage 4), according to the provisional assignments. Since many of these assignments are later overridden the workload imbalance measured by the steering logic suffers a loss of accuracy that translates into an imbalance increase. However, we found that by simply tuning the imbalance threshold used in the steering heuristic, the optimal trade-off between communication and imbalance is restored with almost no performance loss. In more detail, the optimal imbalance thresholds were found experimentally to be 32 and 16 for steering logics with updated and partially outdated information, respectively.

7.2.5 Clustering the Rename Logic (stage 6)

Parallelizing the rename logic is not a trivial task. Advanced techniques for parallelizing the rename logic have been proposed elsewhere [66]. In this work we assume a simple scheme that replicates the rename table and the free lists in all clusters, and keeps them consistent by making identical allocation and renaming operations for all eight destination registers in every cluster. Although such a replication has an area cost, there is a reduction in the number of read ports of the map table by a factor of four, because only the sources of the local instructions are renamed in each cluster (two instructions in the 4-by-2 microarchitecture we are considering).

However, to correctly rename these source registers, the rename logic must consider their dependences on instructions being renamed in all clusters. In addition, to maintain consistent copies of the free lists and rename tables in all clusters, the rename logic in each cluster must know the final assignments of the destination registers of all eight instructions. Hence, the dependence check logic and the overriding muxes in stage 5 (see figure 7-7) must be replicated

to produce identical results in all clusters. Note that some of the dependence check results are used to drive the output multiplexers of both the assignment override and the rename logic.

The rename logic in each cluster renames the local instructions (two instructions in the 4-by-2 microarchitecture we are considering), but for correct operation it must check dependences against instructions being renamed in all clusters. Therefore, the free-list and the rename table are replicated in all clusters and they are kept consistent by making identical allocation and renaming operations for all eight destination registers in every cluster. Although such a replication has an area cost, there is a reduction in the number of read ports of the map table by a factor of four, because only the sources of the two local instructions are renamed in each cluster.

7.2.6 Dispatch (stage 7)

In this stage, the renamed instructions are inserted into the appropriate issue queues in the back-end. We assume this operation takes one cycle, because it involves communication from any cluster to any issue queue, which is comparable to other inter-cluster communications.

The dispatch logic steers instructions from any front-end cluster to any back-end cluster. Each cycle, eight regular instructions plus required copies may be generated and steered. To avoid excessive hardware complexity at this point we constrain each issue queue to receive at most eight instructions per cycle. An issue queue may be the target for more than eight instructions only if one regular instruction requires two copies and both are sent to the same cluster. In this case, the pipeline is stalled for one cycle and the two copies are dispatched in consecutive cycles. However, due to the dependence-based nature of the proposed steering heuristic, this case occurs very rarely. We have experimentally observed less than two cases per million cycles.

The renamed instructions are also inserted into the Reorder Buffer (ROB) during the dispatch stage. The ROB is distributed in the following way. Each cluster has a local ROB that is managed as a FIFO queue. After renaming, two instructions from each cluster are inserted into their corresponding local ROBs. To simplify in-order instruction commit, insertions in all four buffers are coordinated, i.e. 2 entries in each buffer are allocated to every block of instructions fetched in the same cycle, even if the group has fewer than 8 instructions (in this case, some entries are left empty).

The clustered reorder buffer (ROB) must accommodate register copy instructions that may get generated during renaming. Because a copy instruction shares most of its entry fields with its source instruction, an implementation of the ROB could save space by storing the (possible) copy and the source instruction in a single extended entry with optional fields.

7.2.7 Back-End Timing and Commit

We complete our description of the baseline back-end with a summary of the assumed pipeline timing to be used during performance evaluation in the next section. We assume that Instruction Issue, Register Read, ALU Execute, and Writeback are similar to those of a conventional superscalar architecture and take one cycle each.

Instruction commit is assumed to take two pipeline stages as follows. Completed instructions commit in-order as they reach the head of their ROB. To maintain synchronized ROBs, up to eight instructions at the two head entries of all four ROBs must be able to commit at once. To ensure that all eight instructions are complete, the four clusters broadcast the *done* status bits of their two head ROB entries to the other clusters, which takes an additional cycle. This has little impact on performance as it is out of the critical path.

7.3 Microarchitecture Evaluation

In this section, we evaluate the efficiency of the proposed eight-issue four-clustered front-end by comparing it with a baseline processor that uses a conventional centralized front-end to drive the same four-clustered back-end. Note that a strict performance comparison is difficult because the primary motivation for a clustered microarchitecture is to manage on-chip wire delays and permit a very fast clock cycle in future chip technologies. Given the cycle-level simulation being performed, we are only able to measure performance as instructions per cycle (IPC). Hence, the expected advantage in cycle time is not measured. In particular, our IPC performance goal becomes one of matching the performance of using a centralized front-end.

For the experiments in this chapter, we simulated the SpecInt95 benchmark suite, as described in chapter 2. The branch predictor is assumed to operate in a single clock cycle, so we simulate a moderately sized gshare/bimodal hybrid predictor. The bimodal has a 2K-entry table of 2-bit counters; the gshare has a 16K-entry PHT table; and the meta predictor has a 2K-entry table of 2-bit counters. The rest of the architectural parameters are summarized in table 2-1.

7.3.1 Performance

Figure 7-8 compares performance, measured as committed instructions per cycle (IPC) of the clustered front-end microarchitecture to that of the baseline microarchitecture. The graph shows that the IPC of the clustered approach is, on average, 4% lower than that of the baseline. This result is quite uniform for all benchmarks, and has three causes: about 1% is due to switching the predictor banks, another 2.7% is caused by the one-cycle increase of branch misprediction penalty, due to the additional assignment broadcast stage (labelled Dep in figure 7-5), and the rest is caused by the partitioned steering.

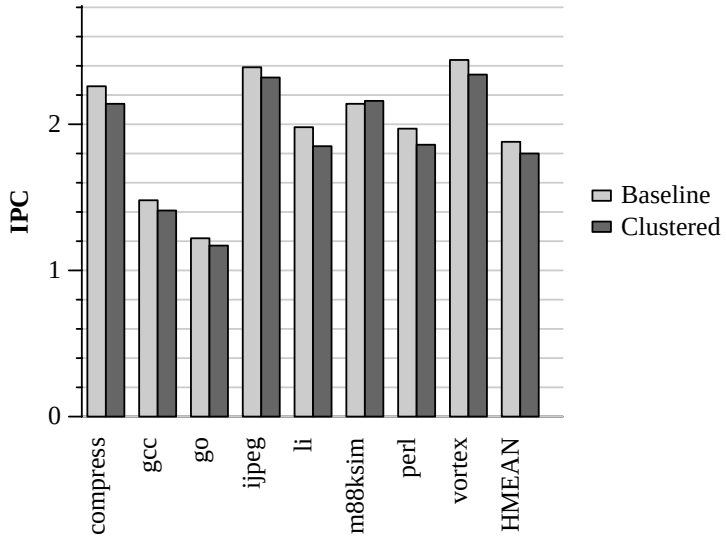

Figure 7-8: IPC of the baseline and the clustered front-end schemes

As discussed in section 7.2.1, the delay of a centralized predictor is the same than a clustered one with four times its size. As an experiment to at least partially account for the clock cycle advantages of the distributed front-end, we performed a second set of simulations where the centralized branch predictor was assumed to have the same size as the predictor in each of the four clusters (hence the distributed predictor is four times as large).

In addition, all the processing tasks between Instruction Fetch and Dispatch are the same for both architectures. The fact that the clustered one has an additional stage does not mean that it has more levels of logic between these two pipeline points, but these levels of logic are spread differently among stages. Therefore, in these experiments it would be fair that if we assume the same cycle time in both cases, we assume also that both architectures have the same latency between these two pipeline stages, by adding one stage to the centralized front-end. Of course, the resulting IPC has not a direct meaning, but it is rather the speed-up the significant metric to be used in this case. When this is done, the clustered front-end shows a 1% average IPC speed-up over the centralized one. Although the speed-up is small, we believe that we have been quite conservative, and we have not factored in all the advantages of the clustered design.

7.3.2 Impact of Partitioning the Branch Predictor

Partitioning the branch predictor produces a one-cycle pipeline bubble each time the cluster that makes the prediction changes. On average, this occurs once every 25 committed instructions for SpecInt95. We evaluated the performance impact of the bank switch penalty on a clustered front-end architecture by comparing it to a similar model with a centralized predictor. It is shown in Figure 7-9 that the one-cycle penalty associated with switching the predictor cluster produces an average performance loss of 1%. The two predictors have identical total size but the clustered one is faster. Alternatively, we could compare predictors having equal latency, by making the predictor in each cluster the same size as the centralized one. The results of this experiment showed a 2.4% average performance speed-up for the clustered predictor.

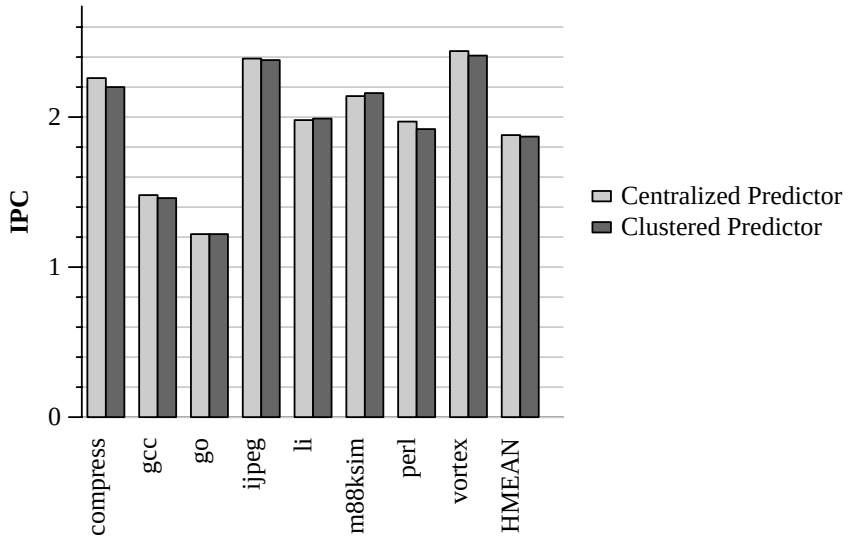

Figure 7-9: Impact of the predictor bank switch penalty. A clustered predictor is compared to a centralized one.

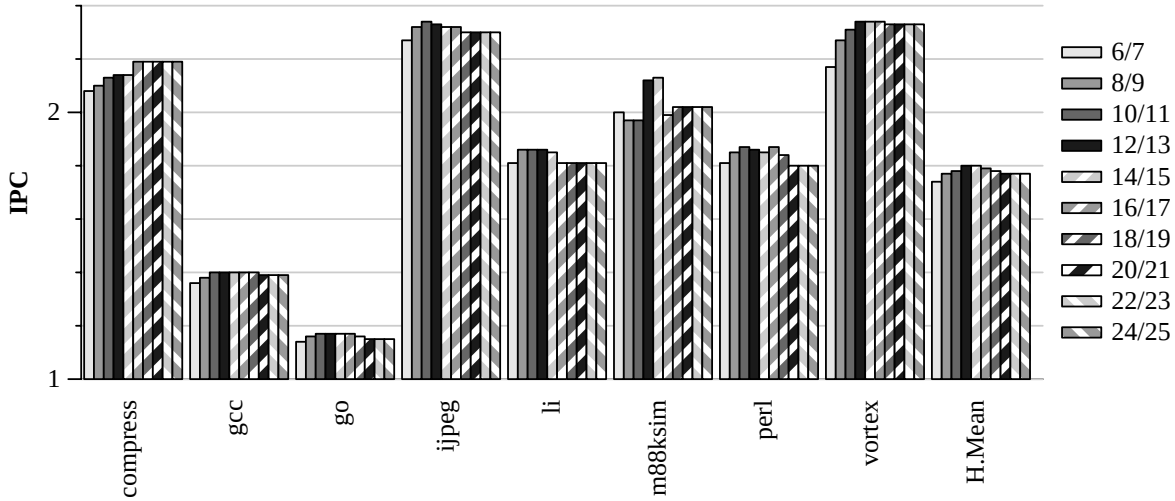

Figure 7-10: Performance impact of the interleaving bits, for a clustered front-end

The interleaving scheme of the default clustered predictor assumed for most experiments (see figure 7-3) interleaves the predictor by the high order bits of the fetch address used to build the index (i.e., the index uses bits 2 to 15, and the default scheme interleaves by bits 14/15). As noted earlier, interleaving by even higher order bits may reduce the amount of bank switching but also may increase aliasing because it tends to concentrate most accesses on the same predictor bank. Conversely, interleaving by lower order bits has the opposite effects. The impact of the predictor interleaving scheme on performance is analyzed in figure 7-10. It is shown that the optimal interleaving differs from one benchmark to another but, overall it stays between bits 12/13 and 16/17, and average performance is maximum for our default interleaving scheme.

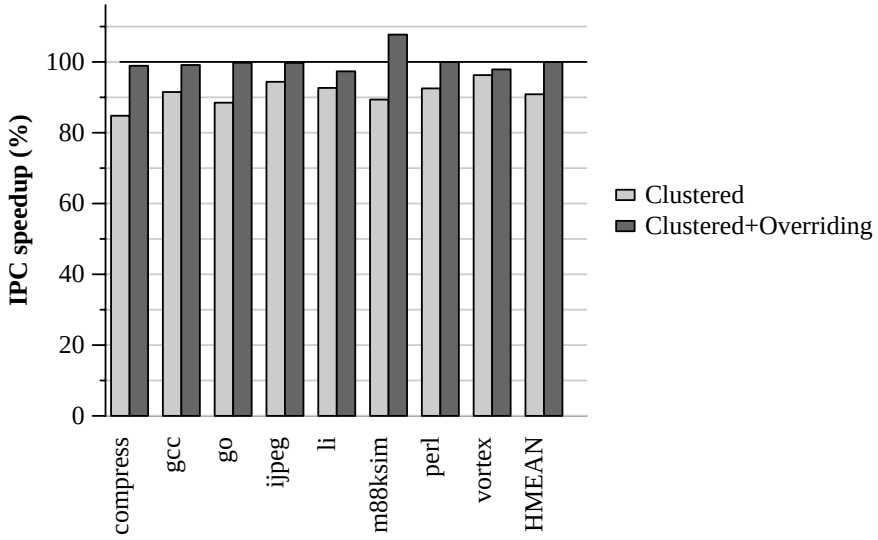

Figure 7-11: Impact of using 2-cycle outdated renaming information by the steering heuristic: two clustered front-ends are compared, a naive scheme and our approach with assignment overriding (speed-ups are relative to a clustered front-end with replicated steering logic that uses updated information)

7.3.3 Impact of Steering with Outdated Renaming Information

Distributing the register renaming function forces steering decisions to be made two cycles before renaming, to have time to broadcast them to all clusters. We evaluated the impact of using outdated renaming information and the effectiveness of the assignment overriding technique by comparing two clustered front-end microarchitectures, with and without overriding, to a model with replicated steering logic that uses totally up-to-date information.

Figure 7-11 shows that using outdated information with the naive scheme, without overriding, produces a 9% average performance loss. Performance loss is due to a drastic increase in inter-cluster data communications between dependent instructions (from 0.18 to 0.43 communications per instruction). However, in our approach with assignment overriding, inter-cluster data communications fall to 0.18 communications per instruction, and the partitioned steering scheme has the same performance as the replicated scheme.

7.4 Conclusions

In this paper, a novel design to fully distribute the processor's front-end is proposed, which extends the advantages of clustering to structures like the I-cache, the branch predictor, the steering logic and the renaming map table.

Several techniques are proposed to partition these structures with the goal of reducing their complexity, and avoiding replication. These techniques minimize the wire delay penalties

caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation, and the cluster assignment.

First, it is shown that the branch predictor latency, which is in the critical path of the fetch address generation loop, may be reduced by partitioning it into clusters, in such a way that cross-structure wire delays are left out of the critical path and converted to cross-cluster communications that may be smoothly pipelined.

Second, it is shown that the latency of a dependence-based steering scheme, which is inherently a serial process that forms a critical hardware loop, may be greatly reduced by partitioning it into clusters. The negative impact of using outdated assignment information is effectively mitigated with an overriding scheme driven by the dependence analysis.

In summary, clustering reduces the latency of these hardware structures and expose wire delays so that they can be dealt with effectively, which results in faster clock rates and translates into significant performance improvements. The schemes proposed in this paper to deal with inter-cluster communications are very effective since they reduce communication penalties to just a 4% IPC degradation, for SpecInt95.