CHAPTER 3

# SLICE-BASED DYNAMIC CLUSTER ASSIGNMENT MECHANISMS

Our first contribution to the design of a clustered architecture is a study of dependence-based partitioning mechanisms that dynamically assign instructions to clusters. Such techniques try to minimize the performance degradation caused by inter-cluster communications and workload imbalance. Inter-cluster communication occurs between dependent instructions that are assigned to different clusters, and it prevents them from executing in consecutive cycles due to the latency of the communication. Workload imbalance appears as a consequence of having local resources, and it may prevent a ready instruction from being issued due to the lack of available functional units in its cluster, even though other idle functional units may exist in other clusters. Both kind of delays negatively impact performance if they affect to instructions in the critical path of execution.

In this chapter, we propose and analyze several dynamic cluster assignment schemes which are based on the concept of slices. Slice-based schemes assign the set of instructions involved in a load or store address calculation (Ld/St slice), or in a branch condition calculation (Br slice), to the same cluster, because these instructions are likely to belong to the critical path. For these experiments, we focused on a cost-effective two-cluster architecture (proposed by Palacharla and Smith [68]), although the proposed schemes can also be used in a generic clustered architecture with symmetric clusters. We show that the proposed dynamic slice-based schemes are more efficient than the static one proposed by Sastry, Palacharla and Smith [87], because they adapt better to the run-time conditions and because they address more effectively the workload balance problem. We also show that, with the proposed General Balance scheme, the cost-effective clustered architecture achieves average speed-ups of 35% over a superscalar with similar complexity, while it only achieved average speed-ups of 4% with the static scheme [87].

## 3.1 Main goals of a Cluster Assignment Mechanism

A clustered architecture exploits the higher simplicity of its components to reduce energy consumption and to permit a faster clock, which are important factors to build high performance computers. The design of the cluster assignment mechanism that distributes the dynamic instruction stream is key for performance in these architectures. In this section we define the two main goals of a cluster assignment mechanism, and we discuss how they are addressed in our proposals, from a conceptual standpoint.

First, due to the partitioning, communications among instructions cannot always use the fast local bypasses, but sometimes they communicate through the slower global paths which may delay the execution of dependent instructions. The use of global bypass paths only in a few cases is an advantage of the clustered approach, since in a centralized architecture all the bypasses are global. Therefore, a primary goal for a cluster assignment algorithm is to minimize penalties caused by inter-cluster communications. Communication delays may degrade performance if the delayed instructions belong to the critical path. In other words, the partitioning goal is to try to execute in the same cluster dependent instructions that belong to a critical path.

Second, due to the partitioning, a clustered processor is less flexible than a centralized one to achieve a uniform utilization of the execution resources. This feature may cause a loss of performance since the amount of execution resources is limited. Thus, it may happen that an instruction in the critical path is delayed by a lack of available resources in its cluster, even though there exists idle resources in other clusters. This additional delay would have been avoided if the steering logic had sent some instructions to a different cluster. We refer to this situation as a workload imbalance among clusters, and since it may potentially degrade the performance, a major goal of the steering logic is to prevent it from happening.

Intuitively, the two main goals, minimal inter-cluster communication penalty and maximal workload balance, are contradictory by nature. Let us just observe that the simplest method for completely eliminating the communications among clusters consists on assigning all the instructions to the same cluster, which is the worst solution for the workload balance goal. Therefore, the task of partitioning involves a trade-off between the two goals that maximizes processor performance. We outline below how these two issues are addressed by the steering logic from a conceptual standpoint. Particular steering techniques are defined in section 3.3.

### 3.1.1 Communication

The task of minimizing penalties by keeping critical communications local to the clusters requires that the instructions in the critical path are identified, which is a very complex task in the context of a dynamically scheduled processor [31, 107]. Therefore, all the known approaches use some heuristics that target a simpler goal.

The slice-based algorithms are based on the observation that load/store instructions and/or conditional branches are likely to belong to the critical path, because these instructions may suffer long delays caused by cache misses and branch mispredictions, respectively. Thus, a goal of the proposed steering scheme is to assign the chain of instructions involved in every load address and/or branch condition calculations to the same cluster.

### 3.1.2 Workload Balance

Obtaining an optimal partitioning that minimizes the delays of critical instructions caused by workload imbalance is a hard problem, due to the difficulty of determining which instructions belong to the critical path, and which will be the availability of functional units when they become ready. The problem is even more complex due to the lack of a well established metric for the intuitive concept of workload balance. All the existing algorithms, as well as those proposed here resort to heuristics to address this problem. In any case, workload balancing should be performed with minimal impact on the communication overhead.

A first naive approach is a random assignment to either cluster where all clusters have the same probability of being selected. A second approach detects when there is a workload imbalance and how much unbalanced it is, and also determines which is the least loaded cluster. There are many alternatives to determine at run-time the individual workloads of the clusters and their relative imbalance, because there is not a unique definition.

The workload imbalance may be estimated by counting the difference in the number of instructions steered to each of the two clusters (we refer to this metric as I1). However, this metric does not consider the amount of parallelism present in each instruction window at a given time. On the other hand, the workload of a cluster may be computed as the number of ready instructions it has. The workload is considered imbalanced when one cluster has more ready instructions than its issue width, and the other has less than its issue width. Just in this scenario, the instant workload imbalance is quantified as the difference in number of ready instructions (metric I2). In any other scenario, the processor can execute the instructions at the maximum possible rate, so the workload is then considered balanced.

The load balancing mechanism presented in this chapter considers the two metrics (I1 and I2) by maintaining a single integer imbalance counter that combines the two informations. Each cycle, this counter is updated by adding the I1 metric to the average of I2 with the previous values of the counter along the last N cycles (we have determined empirically that 16 is an adequate value for N).

Although the schemes presented here use the above combination of I1 and I2, we have empirically observed that the metric I1 is more effective than the I2 to balance the workload when both are considered isolated. In fact, since metric I1 alone gives performance figures quite close to those produced by the combination of I1 and I2, it could be used alone in a particular cost-effective implementation. On the other hand, since I2 matches more closely the concept of imbalance as described in the beginning of section 3.1, I2 will be used to report the workload imbalance of the experiments.

## 3.2 The Cost-Effective Clustered Microarchitecture

The slice-based code partitioning schemes presented and evaluated in this chapter are built upon the early proposal of Palacharla and Smith [68], also developed in collaboration with Sastry [87]. In those papers it is described how a typical superscalar with an integer and a FP subsystem may benefit from augmenting the latter with a few ALUs capable of executing simple integer operations. Since exploiting the new features requires that some integer instructions are steered to the FP subsystem, they propose a compile-time cluster assignment method based on the concept of "program slices" [87]. In order to facilitate comparisons, our proposals are evaluated with a similar architecture, although they could be applied to any other clustered architecture. This architecture will be referred to as the cost-effective clustered architecture, and is briefly described below.

The motivation for the cost-effective clustered microarchitecture starts from the observation that many current superscalar processors are already partitioned into two subsystems or clusters, the integer and the floating point one, but the whole FP subsystem remains idle during the execution of integer programs. Therefore, the FP subsystem can be easily extended to execute simple integer and logical operations (no multiplication and division), which represents a very small added hardware cost considering today's transistor budgets. The advantage of the new architecture is that its floating-point registers, data path and mainly, its issue logic are used for any type of application.

The cost-effective clustered architecture we used to conduct our experiments is very similar to the one mentioned above (see figure 3-1). It consists of two clusters, each containing a register file, an issue queue, and some basic integer and logic functional units. Moreover, one of them, which will be referred to as the integer cluster, contains also complex integer functional units (multiplier and divider), while the other cluster, referred to as the FP cluster, contains also the floating-point functional units. There is also a communication datapath to copy values from one cluster to the other. Loads and stores are divided into two operations. One of them calculates the effective-address and the other accesses memory. The effective-address is computed in an adder, in either cluster, and then it is forwarded to the access instruction, in the common disambiguation hardware. A load access is issued when a memory port is available and all prior stores know their effective address. If the effective address of a load matches the address of a previous store, the store value is forwarded to the load. Store accesses are issued at commit time.

Our cost-effective clustered architecture differs from the one proposed by Palacharla et al. [68] in the following aspects. First, since our steering schemes are dynamic, there is a piece of hardware, referred to as the steering logic, to perform the cluster assignments and monitoring the workload of clusters. Second, the register rename table needs to be extended to support up to two mappings per logical register, one for each cluster. Third, load and store address calculations may execute in either cluster, which removes one important constraint of the previous proposal.

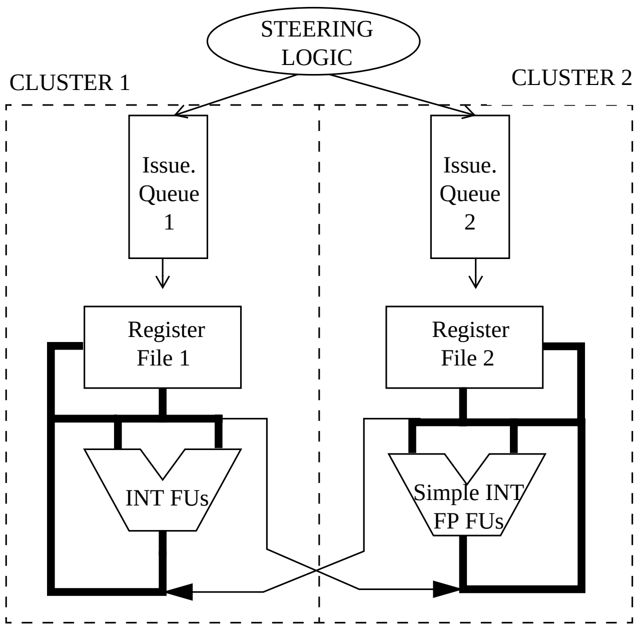


**Figure 3-1: Block diagram of a cost-effective clustered architecture**

Finally, looking at the above description, one realizes that our cost-effective clustered architecture is almost just a particular instance of our Reference Clustered Architecture with the only difference that the two clusters are heterogeneous. Since all the rest of mechanisms that support the clustered approach - map table extensions, copy instructions, inter-cluster bypasses, etc - are identical to those of the Reference Clustered Architecture, we omit here a full description (see chapter 2 for details).

## 3.3 Cluster Assignment Schemes

This section presents several new cluster assignment schemes and evaluates their performance. We first define some terminology and describe the experimental framework. Then, we compare the effectiveness of a static assignment versus a simple dynamic mechanism. Finally, other alternative dynamic schemes are presented and evaluated.

### 3.3.1 Terminology

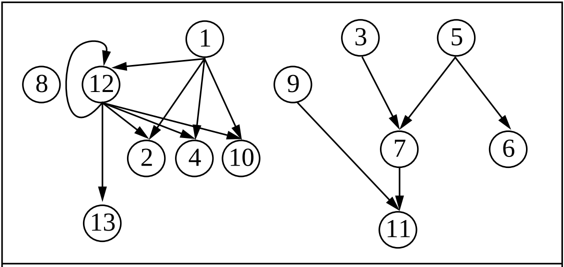
A register dependence graph (RDG) represents all register dependences in a program. It is a directed graph that has a node associated to each static instruction and an edge for every data dependence (true dependence) through a register. Memory instructions are special cases since they are split into two disconnected nodes, one representing the address calculation and the other the memory access. Figure 3-2 shows an example of an RDG. Note that for the sake of clarity, in the assembly code, memory instructions have already been split into two, one for address calculation (EA) and another for the memory access LD/ST.

```
for (i=0;i<N;i++) {
     if (C[i]!=0)
          A[i]=B[i]/C[i];
     else A[i]=0;
}
```

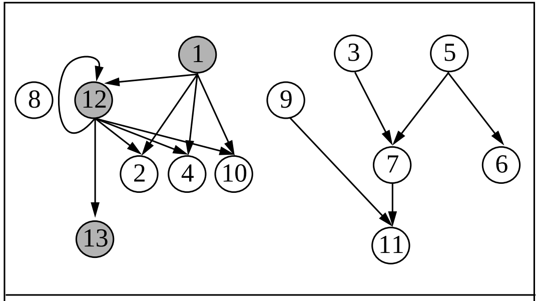
Code in C

```
1               MOV   0, Ri
2    for:       EA    RB+Ri
3               LD    RBi
4               EA    RC+Ri
5               LD    RCi
6               BEQZ  RCi, l1
7               DIV   RBi, RCi, RAi
8               JMP   l2
9    l1:        MOV   0, RAi
10   l2:        EA    RA+Ri
11              ST    RAi
12              ADD   Ri, 4, Ri
13              BNEQ  Ri, N*4, for
```

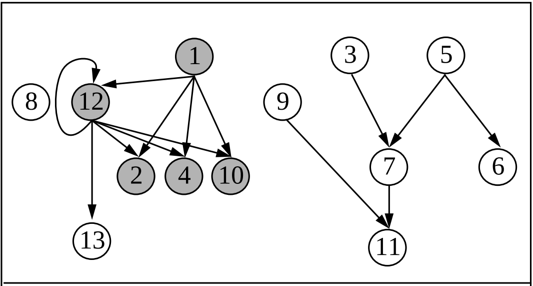
Assembly code (R*A*, R*B* and R*C* contain the initial addresses of arrays A, B, C respectively)



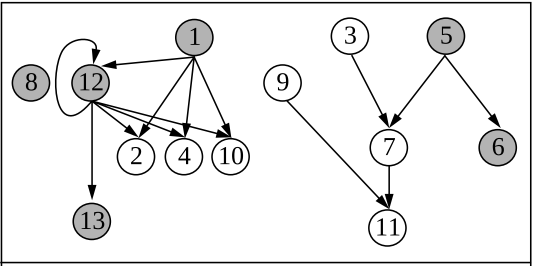
Register dependency graph



Backward slice with respect to node 13 (shaded nodes)



LdSt slice (shaded nodes)



Br slice (shaded nodes)

**Figure 3-2: Example of a RDG**

The *backward slice* of an RDG with respect to a node *v* is defined as the set of nodes from which *v* can be reached, including *v* [87]. Figure 3-11 shows the backward slice with respect to node 13 of the sample RDG.

The *LdSt slice* of a program is defined as the set of all instructions that belong to a backward slice of any address calculation instruction. Similarly, the *Br slice* of a program consists of all instructions that belong to the backward slice of any branch instruction. Figure 3-2 shows the LdSt slice and the Br slice of the sample program.

### 3.3.2 Experimental Framework

Performance figures were obtained through a cycle-accurate timing simulator based on the SimpleScalar tool set v3.0 [15], which was extended to simulate the architecture described in section 3.2. Results are presented for the SpecInt95 benchmark suite. Table 3-1 lists the benchmark programs and their inputs. Programs were compiled with the Compaq/Alpha C compiler with the -O5 optimization flag. For each benchmark, 100 million instructions were run after skipping the first 100 million. Table 3-2 shows the architectural parameters of the assumed processor.

| **Benchmark** | **go** | **li** | **gcc** | **compress** | **m88ksim** | **vortex** | **ijpeg** | **perl** |
|---|---|---|---|---|---|---|---|---|
| Input | bigtest.in | *.lsp | insn-recog.i | 50000 e 2231 | ctl.raw, dcrand.lit | vortex.raw | pengin.ppm | primes.pl |

**Table 3-1: Benchmarks and their inputs**

Performance will usually be reported as speed-up over a *base architecture*, which is a conventional microprocessor with the same architectural parameters listed in Table 3-2 except that it has neither integer units in the FP cluster nor inter-cluster bypasses.

| **Parameter** | **Configuration** | |
|---|---|---|
| Fetch width | 8 instructions | |
| I-cache | 64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty | |
| Branch Predictor | Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters. | |
| Decode/Rename width | 8 instructions | |
| Instruction queue size | 64 | 64 |
| Max. in-flight instructions | 64 | |
| Retire width | 8 instructions | |
| Functional units | 3 int ALU + 1 int mul/div | 3 int ALU + 3 fp ALU + 1 fp mul/div |
| | 3 comm/cycle to C2 | 3 comm/cycle to C |
| | Communications consume issue width | |
| Issue mechanism | 4 instructions | 4 instructions |
| | Out-of-order issue<br>Loads may execute when prior store addresses are known | |
| Physical registers | 96 | 96 |
| D-cache L1 | 64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty | |
| | 3 R/W ports | |
| I/D-cache L2 | 256 KB, 4-way set associative, 64-byte lines, 6-cycle hit time. | |
| | 16 bytes bus bandwidth to main memory, 16 cycles first chunk, 2 cycles interchunk. | |

**Table 3-2: Machine parameters (split into cluster 1 and cluster 2 if not common)**

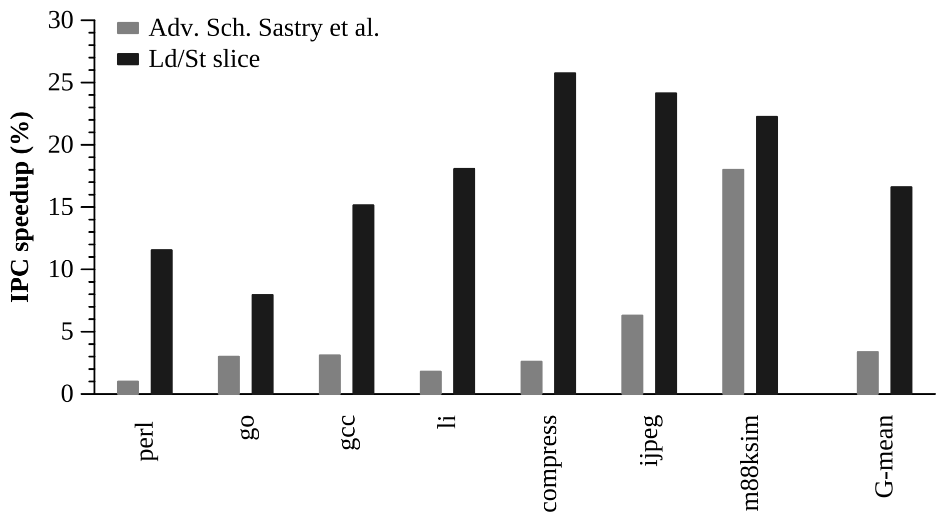


**Figure 3-3: Static versus dynamic LdSt slice steering**

## 3.3.3 Static versus Dynamic Cluster Assignment with LdSt Slice Steering

Cluster assignment can be done at compile time (static) or at run time (dynamic). The first method relies on the compiler, which allocates each static instruction to a cluster, while the second method is based on a specialized hardware that decides where to dispatch each dynamic instruction. The main advantage of a static assignment is that it requires minimal hardware support, but its downside is that it requires to recompile the applications because it extends the ISA for encoding the steering information. Furthermore, recompilation is needed for each new microarchitecture generation that changes the relevant features of the clusters.

In contrast, a dynamic assignment method does not require to recompile, because it makes clustering transparent to the compiler. In addition, the information used by the dynamic steering logic (workload balance, data dependences) is obtained directly from the actual pipeline state, rather than estimations of the compiler. Therefore, a dynamic steering scheme is more effective than a static approach because it is more adaptable to the actual processor state. This work focuses on this type of steering.

The static assignment proposed by Sastry *et al.* [87] is based on sending all instructions that belong to the subgraph defined by the LdSt slice, probably extended with neighbor instructions, to the integer cluster. This extension is based on some heuristics that try to approximate its effect in terms of workload balance and communication overheads.

We have evaluated the speed-ups of the cost-effective architecture over a conventional architecture (one without the simple integer units added to the FP cluster). Figure 3-3 compares the speed-ups of Sastry’s et al. static assignment with the speed-ups achieved by a simple dynamic assignment that tries to dispatch all instructions in the LdSt slice to the integer cluster and the remaining instructions to the FP cluster (excepting complex integer instructions). We will refer to this dynamic assignment scheme as *LdSt slice steering*. This dynamic assignment can be implemented by including a table that is indexed by the PC of instructions. For each entry it has a one-bit flag that denotes whether the corresponding instruction belongs to the LdSt slice or not. Initially all the bits are cleared. For every instruction, if it is a memory instruction its flag is set. If an instruction finds its flag set, the flags of its parents in the RDG are also set. The parents are identified by means of an additional table that holds for each logical register the PC of the last decoded instruction that uses it as a destination register.

For the experiments in figure 3-3, the numbers for the static assignment have been obtained from the original paper [87] and the dynamic approach has been simulated using the same compiler, the same compiler options, the same benchmarks and the same architecture. Note that the dynamic scheme significantly outperforms the static one for all the programs excepting m88ksim, for which both schemes achieve similar levels of performance. On average, the *LdSt slice steering* achieves an speed-up of 16% whereas the static assignment speed-up is just 3%.

### 3.3.4 LdSt Slice Steering versus Br Slice Steering

The performance of any assignment scheme is quite sensitive to the number of inter-cluster communications that it generates. A communication has some latency that may delay the execution of the consumer instructions. Therefore, the criticality of consumer instructions is even more important than the absolute number of communications. An inter-cluster communication that is consumed by an instruction that is not critical may have no effect on the execution time. Some memory instructions, especially those that cause many cache misses, are critical in most programs, which suggests that the LdSt slice steering may be an appropriate assignment scheme because executing all the backward slice of a load in one cluster avoids adding communication delays to the computation of its address. However, branch instructions are also critical in non-numeric codes such as the SpecInt95. This suggests an alternative assignment scheme that steers instructions in the Br slice to the integer cluster and the remaining instructions to the FP cluster (excepting complex integer instructions). We will refer to this scheme as Br slice steering. The hardware to implement this scheme is basically the same as that described in section 3.3.3 for the LdSt slice steering.

Figure 3-5 compares the performance of the LdSt slice steering with that of the Br slice steering. Note that the performance of the Br slice steering is somewhat lower, which is explained by the larger number of communications that it generates, as shown in Figure 3-4. This figure shows the average number of communications per dynamic instruction, split into critical and non-critical. We consider that a communication is critical when there is any instruction in the destination cluster that has been delayed due to the communication.

Another critical factor for the performance of a clustered architecture is the workload balance. Figure 3-6 shows the distribution function of the I2 workload imbalance metric, i.e., the difference between the number of ready instructions in each cluster for each cycle (see section 3.1.2). It can be seen that both dynamic assignment schemes result in a similar workload balance. In both cases, there is a significant percentage of time in which the two clusters have different workload: either the integer or the FP cluster is overloaded. Note that the overload of the FP cluster could be reduced if some of the instructions that are not part of the LdSt slice

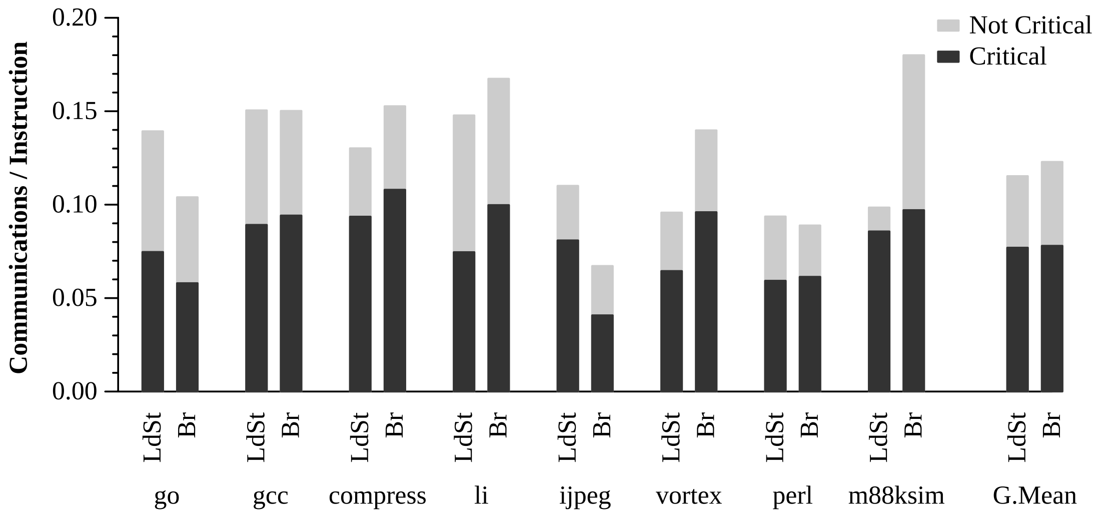


**Figure 3-4: LdSt slice versus Br slice steering: communications per instruction**

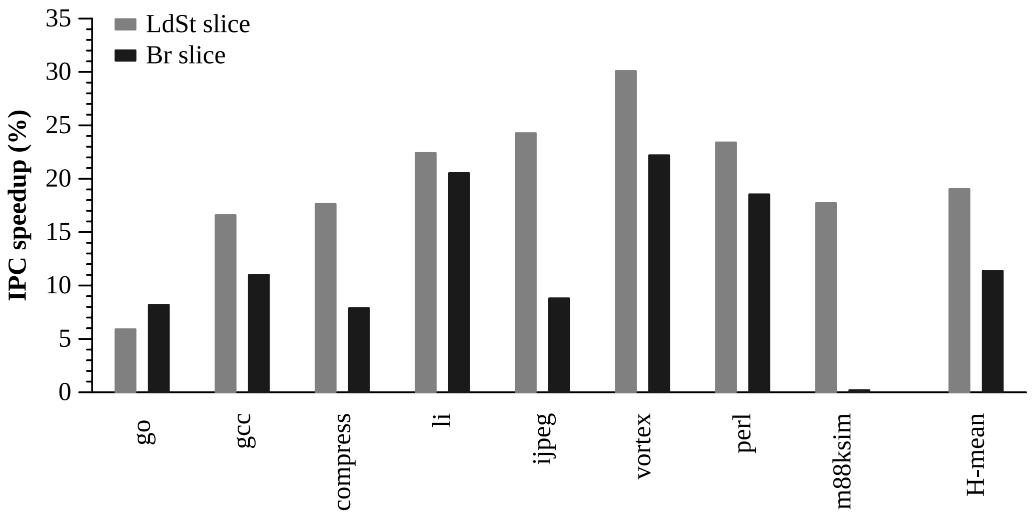


**Figure 3-5: LdSt slice versus Br slice steering: performance**

(resp. Br slice) were dispatched to the integer cluster. This motivates the next assignment scheme.

### 3.3.5 Non-Slice Balance Steering

As motivated in the previous section, a better workload balance could be achieved if instructions that are not in the slice are used to balance the workload. However, sending every non-slice instruction to the least loaded cluster would result in too many communications. A more effective approach would be to send non-slice instructions to the least loaded cluster only when the absolute value of the workload imbalance counter (see section 3.1.2) exceeds a given

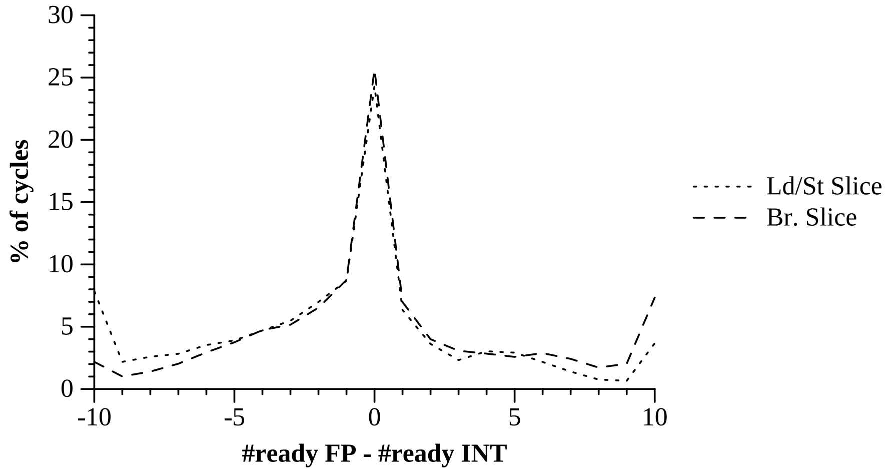


**Figure 3-6: LdSt slice versus Br slice steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)**

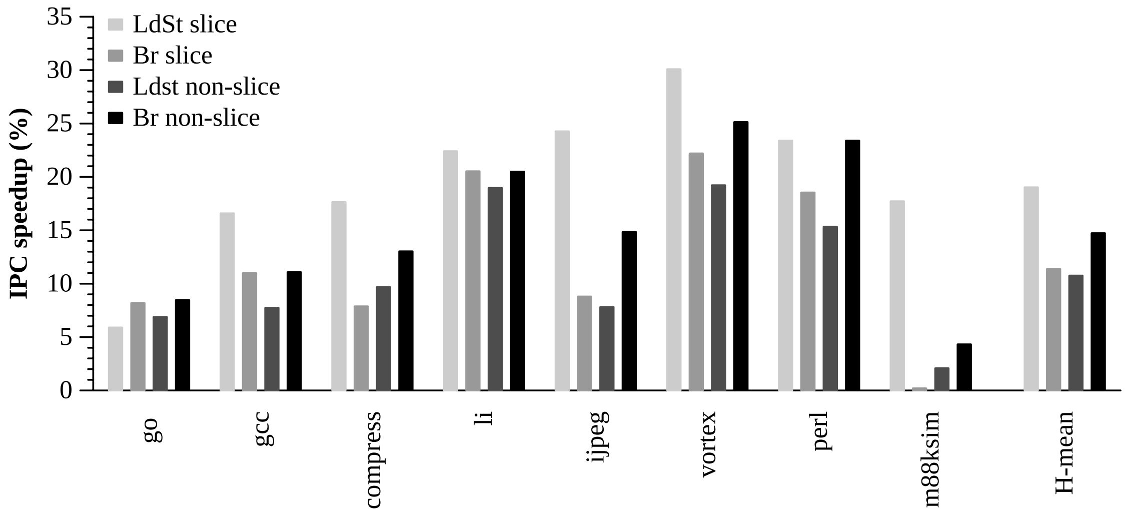


**Figure 3-7: Non-slice Balance steering versus slice steering: performance**

threshold. Otherwise, these instructions are sent to the cluster where their operands reside in order to reduce communications. We refer to this approach as *non-slice balance* steering. We have empirically determined that a threshold of 8 is adequate for the cost-effective architecture.

Figure 3-7 compares the performance of the non-slice balance steering with that of the slice steering. It can be seen that the non-slice balance steering is beneficial for the Br slice but detrimental for the LdSt slice, in spite of the fact that this scheme improves the workload balance. This is explained by the amount of communications that these schemes generate, which are depicted in Figure 3-8. This figure shows that the non-slice balance steering significantly increases the number of communications for the LdSt slice whereas it has about the same number of communications as the slice steering for the Br slice.

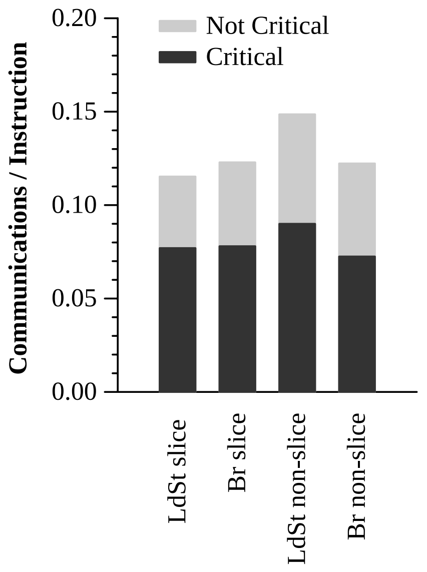


**Figure 3-8: Non-slice Balance steering versus Slice steering: number of communications per dynamic instruction (SpecInt95 average)**

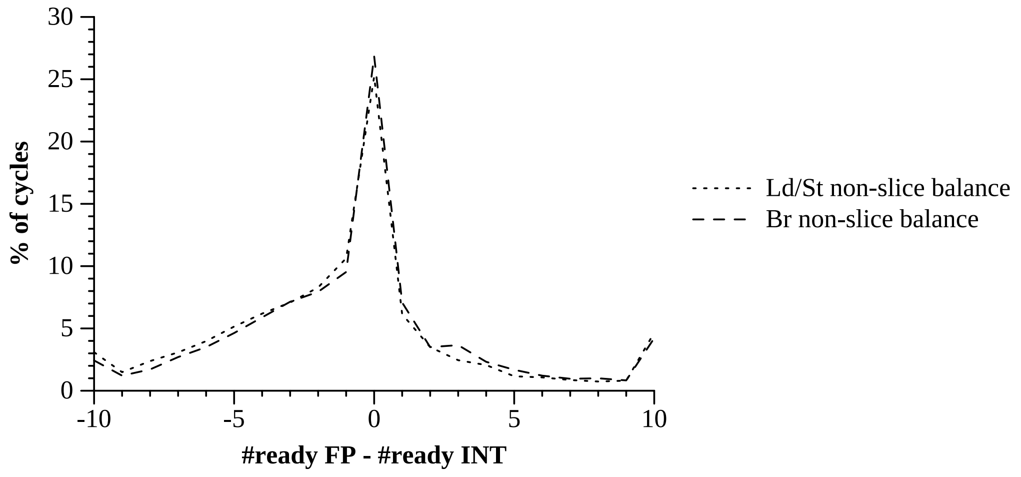


**Figure 3-9: Non-slice Balance steering versus slice steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)**

Figure 3-9 shows the distribution function of the workload balance for the non-slice balance steering. Note that the workload balance has improved (see the shape of the curve) in comparison with the slice steering scheme (figure 3-6), but there is still a large percentage of cycles where the imbalance is significant. It is especially remarkable the overload of the integer cluster, which motivates the next assignment scheme.

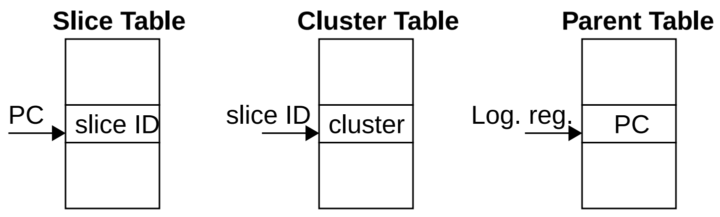


**Figure 3-10: Hardware support for the Slice Balance steering**

### 3.3.6 Slice Balance Steering

The Br slice (or LdSt slice) of a program consists of several backward slices of branches (resp. loads/stores). A better balance could be achieved if the instructions in a given backward slice were sent to the same cluster but different backward slices could be sent to different clusters. We refer to this scheme as slice balance steering.

In this scheme, instructions are classified into backward slices (or slices for short) at run-time by means of the tables shown in Figure 3-10. The slice table identifies for each instruction the slice to which it belongs. The backward slice of instruction *v* is identified by the PC of *v*. Initially no instruction belongs to any slice, which is denoted by a special value in the slice table. When a branch is executed (resp. a load/store), the slice table is modified to indicate that this instruction belongs to its own slice. Every time that an instruction in a slice is executed, it propagates the slice ID to its parents, which are identified by means of the parent table. For each logical register, this table holds the PC of the last decoded instruction that uses this register as its destination operand. The cluster where each slice is currently mapped is identified by means of the cluster table.

Instructions that belong to a slice are dispatched to the cluster where the slice is assigned (according to the cluster table). However, if this cluster is strongly overloaded (using the same workload measures as in the previous steering scheme), the whole slice is re-assigned to the other cluster. Instructions that do not belong to any slice are handled as in the non-slice balance steering approach.

Figure 3-11 shows the speed-up of the slice balance steering scheme over the base architecture. It can be seen that the performance for both types of slices (LdSt and Br) are very similar, and overall, the effectiveness of this approach is much higher than previous schemes. The average speed-up is 27% for the LdSt slice and 26.5% for the Br slice.

This good performance is due to a significant improvement in workload balance and a reduction in number of communications alike. Figure 3-12 shows the distribution of the I2 workload imbalance metric for the slice balance steering (LdSt and Br) and compares it with that of a naive steering scheme that alternatively sends instructions to each cluster, if they can be executed in both. Note that this scheme has a low performance (as we will later show) due to its high number of communications, but it distributes the workload quite evenly. We refer to this scheme as *modulo steering*. We can see that the workload balance of the slice balance steering

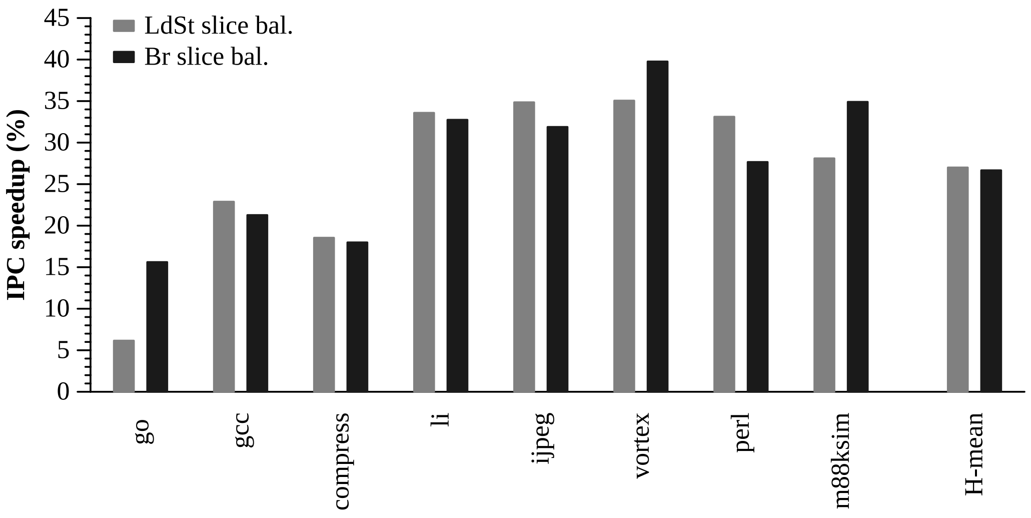


**Figure 3-11: Slice Balance steering: performance**

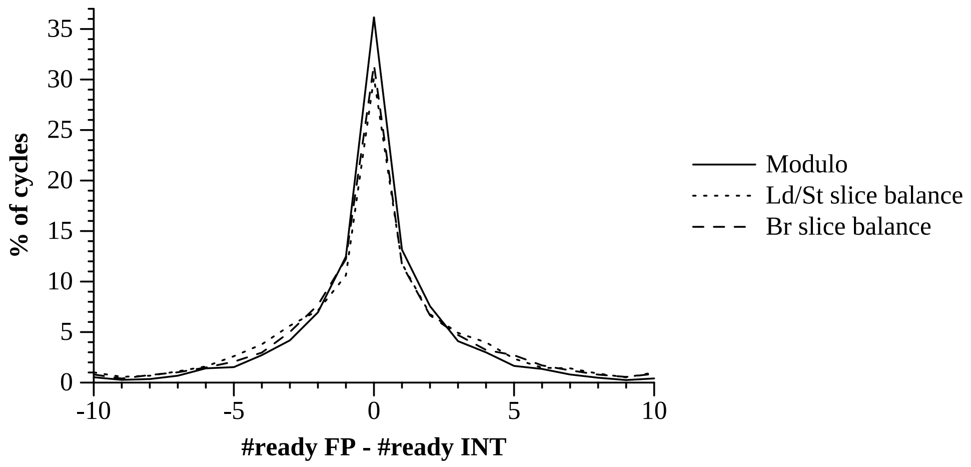


**Figure 3-12: Slice Balance steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)**

is almost the same as that of the modulo steering. Regarding communications, the slice balance steering generates 0.07 (LdSt) and 0.08 (Br) communications per dynamic instruction on average, which is quite less than previous schemes.

### 3.3.7 Priority Slice Balance Steering

The objective of dispatching a whole slice of a load/store or branch instruction to the same cluster is to avoid communications in critical parts of the code. However, not all slices are equally critical. In particular, one may expect that slices corresponding to loads that miss very often in cache, or branches that are wrongly-predicted very often are more critical than the others since they cause significant penalties. Thus, slices could be classified according to their criticality. Computing the criticality of each instruction is by itself a complex problem that is

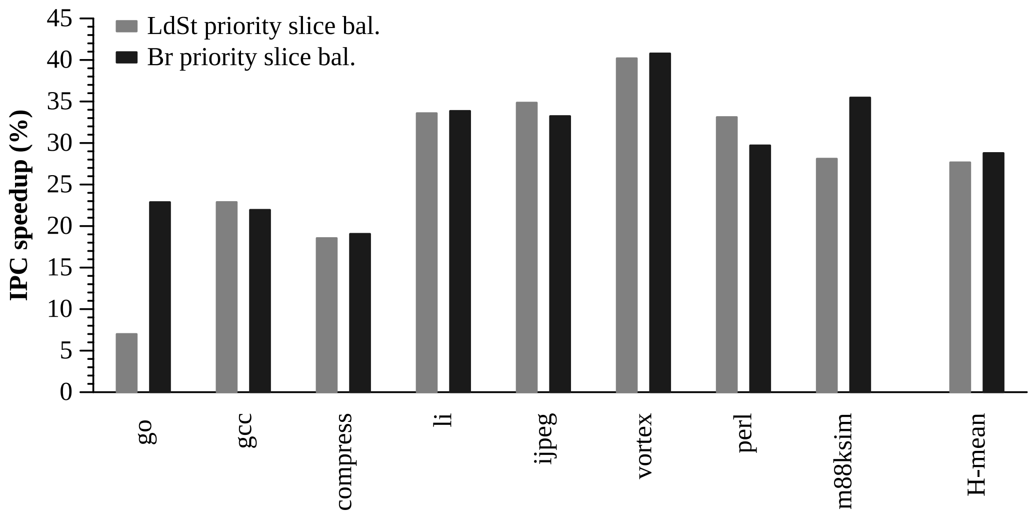


**Figure 3-13: Priority Slice Balance steering: performance**

beyond the scope of this work. Instead, we approximate the criticality of a slice by the number of cache misses or branch mispredictions of the instruction that defines the slice, depending on the type of slice.

The *priority slice balance* steering tries to dispatch the instructions of any slice corresponding to a critical instruction to the same cluster, whereas the remaining instructions are dispatched following the same approach as the *non-slice balance* steering scheme. The threshold for deciding whether an instruction is critical or not will be dynamically adjusted so that around 50% of the instructions belong to critical slices. In particular, every 8192 ($2^{13}$) cycles the processor computes the number of instructions that have been considered as belonging to a critical slice. If this number is higher than half of the number of executed instructions, the threshold is increased; otherwise, it is decreased.

The main advantage of this scheme is that now, only the critical slices will be treated as such. This scheme improves the flexibility for balancing the workload since there are more instructions that are individually treated than in the previous schemes. Having more flexibility to balance the workload with individual instructions reduces the number of slice re-mappings caused by strong imbalances (see section 3.1.2 for a definition). Such re-mappings can arise in the middle of the execution of a given slice, and therefore, they may cause undesired intra-slice communications. Thus, we expect this scheme to reduce the number of critical communications, although it might increase the total number of communications when trying to improve the workload balance. Overall, this scheme tries to minimize the communications in the critical slices while it tries to maximize the workload balance by means of the rest.

As far as the hardware implementation is concerned, we need a cycle counter (13 bit counter), a threshold register with an increment and decrement hardware, a critical instruction counter –16 bits are enough ($2^{13}$cycles x $2^3$ issue-width)– and a non-critical instruction counter. In addition, the cluster table (see figure 3-10) should be augmented with a new field that counts for each slice the number of cache misses or branch mispredictions of the instruction that defines the slice, and a flag that indicates whether the slice is critical.

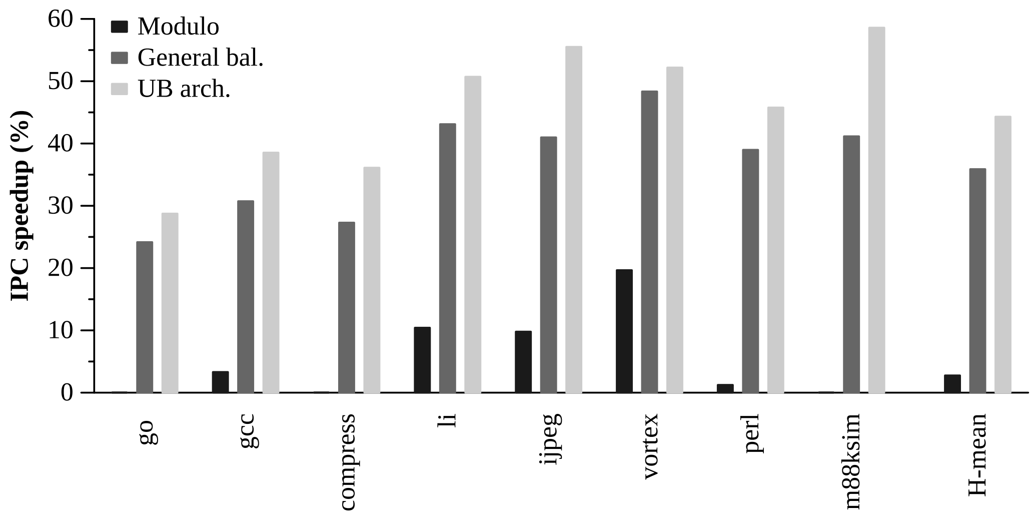


**Figure 3-14: General Balance steering**

Figure 3-13 shows the performance of the priority slice balance steering. It achieves an average speed-up of 27.7% (LdSt slice) and 28.8% (Br slice) over the base architecture, which is slightly better than that of the slice balance steering (see figure 3-11). This improvement is due to the reduction in number of critical communications per dynamic instruction, which on average decreases from 0.050 to 0.045 for the LdSt slice and from 0.055 to 0.043 for the Br slice.

## 3.3.8 General Balance Steering

The *general balance* steering is a particular case of the previous steering scheme, in which the criticality threshold is set so high that there are no critical instructions, all instructions are steered as if they were non-slice instructions. That is, instructions are sent to the least loaded cluster when there is a strong workload imbalance or they have an equal number of operands in both clusters. Otherwise, they are sent to the cluster where most of their operands reside. The immediate consequence is that the required hardware to identify program slices (see figure 3-10) is not needed and no extra hardware is required to detect the criticality of instructions. Actually, the general balance scheme falls into a different category, the instruction-based schemes, rather than slice-based, because it makes all steering decisions at a single instruction granularity. Thus, the performance of this scheme is analyzed here to provide comparison with the rest of the slice-based schemes, but it will be analyzed in more detail in chapter 4 (where it is called the Advanced RMB scheme), in the context of our Reference Cluster Architecture.

Figure 3-14 shows the performance of the *general balance* scheme. It also includes the performance of the *modulo* steering (see section 3.3.6) and that of a conventional 16-way issue processor (8 integer and 8 FP). The performance of this latter architecture can be considered as an *upper-bound* for any instruction assignment approach since it has the same integer instruction throughput as the assumed architecture but it does not incur in any communication penalty. The general balance steering achieves an average speed-up of 36%, which is higher

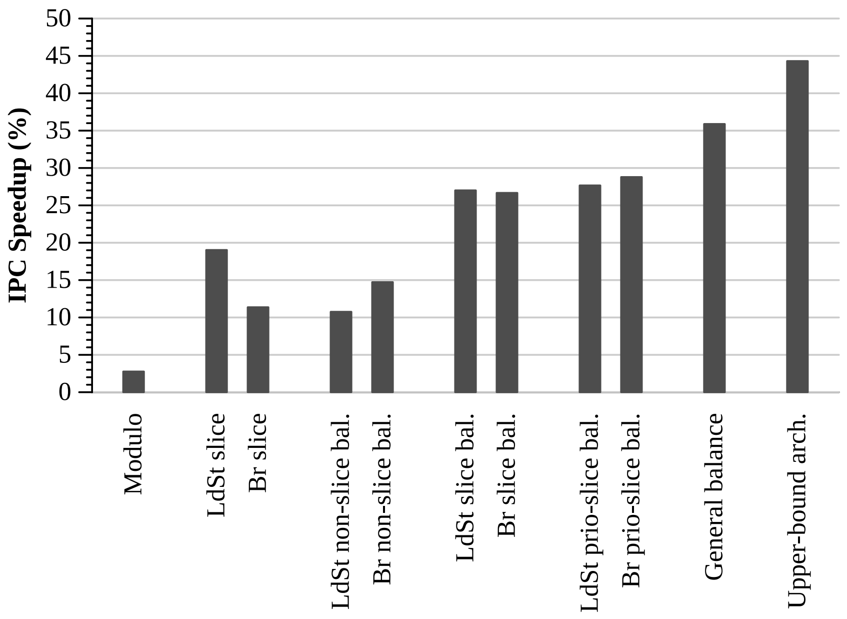


**Figure 3-15: Performance comparison among all the proposed steering schemes**

than previous schemes and just 8% smaller than the upper-bound. On the other hand, the modulo steering produces a rather low improvement (2.8% on average).

## 3.4 Evaluation

In this section it is summarized the performance results of all the above proposed cluster assignment schemes. Next, the best performing scheme is compared to another dynamic steering scheme previously found on the literature, and it is analyzed the suitability of the cost-effective architecture for FP programs.

### 3.4.1 Overall Performance Comparison

Figure 3-15 depicts, for each steering scheme, the SpecInt95 average speedups of the cost-effective architecture over the base architecture. The graph also includes, for comparison, the performance of the *upper bound* architecture (see section 3.3.8). The best slice-based scheme is the priority slice balance, which achieves a speedup of 28%, but the best performance is obtained with the general balance steering scheme, which achieves an average speedup of 36%, just 8% smaller than the upper bound. In other words, this steering scheme enables the cost-effective architecture to realize most of its potential, with integer programs.

These results show that in a cost-effective architecture, with an appropriate dynamic steering like our *general balance* scheme, idle floating-point resources can be profitably exploited to speed-up integer programs, with minimal hardware support and no ISA change.

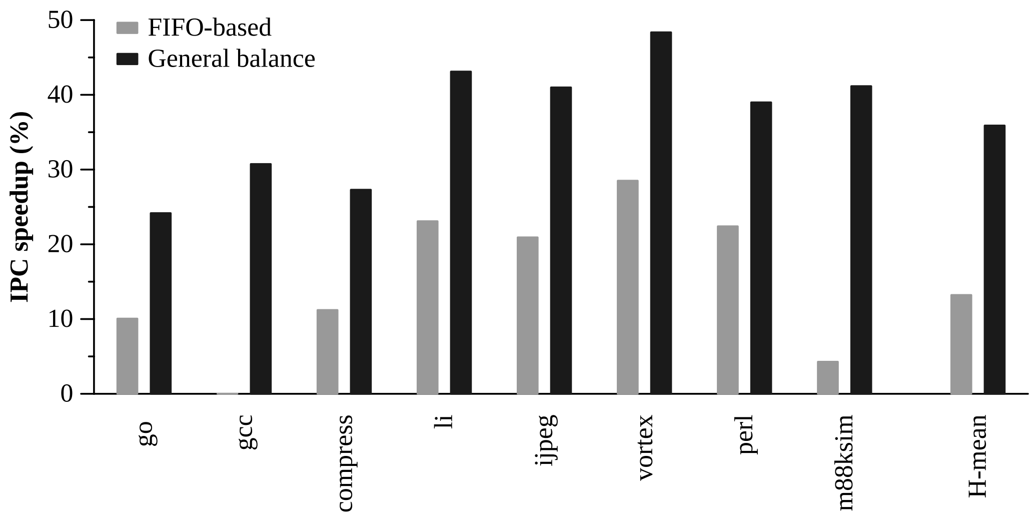


**Figure 3-16: General Balance steering versus FIFO-based steering [70]**

### 3.4.2 Comparison to a Static Slice-Based Scheme

Sastry, Palacharla and Smith [87] proposed a static slice-based assignment algorithm, similar to our dynamic LdSt slice scheme. Both schemes were compared in section 3.3.3 (figure 3-3), and the results showed that the dynamic approach is much more effective than the static one: the speed-ups of the dynamic scheme are up to 10 times bigger than those of the static scheme, for several reasons: first, because a dynamic steering technique does a better job not only at reducing inter-cluster communication but also workload imbalance, which was already reported to be one of the main drawbacks of the static approach [87]; second, because a dynamic steering adapts more accurately to many run-time conditions that are difficult to estimate at compile time.

From these results and those of the previous section we conclude that all the proposed schemes significantly outperform the previous static slice-based proposal.

### 3.4.3 Comparison to Another Dynamic Scheme

Palacharla, Jouppi and Smith [70] proposed a dynamic assignment approach, for a different clustered architecture, that could also be applied to our assumed architecture. Their basic idea is to model each issue queue as if it was a collection of FIFO queues. Instructions are steered to FIFOs following some heuristics that ensures that any two consecutive instructions in a FIFO are always dependent. If all the source registers are ready, or their producers do not stay at any of the FIFO tails, then the instruction is dispatched to an empty FIFO. The assignment policy chooses always empty FIFOs from the same cluster until all of them are used, then it switches to another cluster (for more details refer to the original paper [70]).

For the following experiment, we modelled the FIFO-based steering with 8 FIFOs per cluster, each with 8 entries (thus having a total of 64 scheduler entries per cluster), on our

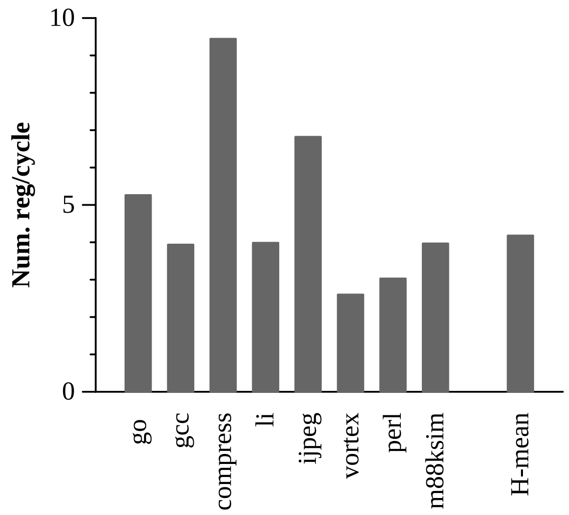


**Figure 3-17: Register replication on a cost-effective 2-clusters architecture**

assumed cost-effective architecture. Figure 3-16 compares the performance of the general balance scheme versus the FIFO-based scheme. The results show that the general balance steering significantly outperforms the steering scheme based on FIFOs for all the programs. On average, the FIFO-based steering increases the IPC of the conventional microarchitecture by 13% whereas the general balance steering achieves a 36% improvement.

This difference in performance is explained by the fact that both schemes result in quite similar workload balance but the FIFO-based approach generates a significantly higher number of communications. On average, the general balance steering produces 0.042 inter-cluster communications per dynamic instruction whereas the FIFO-based approach results in 0.162 communications. Note that the FIFO-based scheme does not usually dispatch an instruction to the cluster that produces its operands if they are ready, or their producers are not at some FIFO tails.

## 3.4.4 Register Replication

As outlined in section 3.2, the cost-effective 2-clusters microarchitecture requires some degree of register replication. We have evaluated the average number of logical registers that have a physical register allocated in both clusters, with the general balance steering scheme. The results in figure 3-17 show that the required register replication is very low. Instead of replicating the whole physical register file, as for instance the Alpha 21264 processor does [53], this architecture requires on average only 3.1 registers to be replicated. This saving in register storage may have a significant impact on the register file access time, which in turn is one of the critical delays of superscalar processors [29, 70].

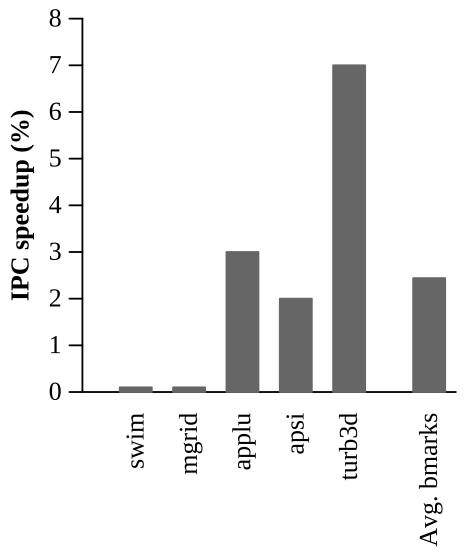


**Figure 3-18: Performance of the SpecFP95 on a cost-effective 2-clusters architecture**

### 3.4.5 Running FP Programs on a Cost-Effective Clustered Architecture

So far, all the evaluations of the cost-effective clustered architecture included only integer benchmarks because the particular optimizations of this architecture specifically target this kind of applications. However, we want to investigate whether sending integer instructions to the FP cluster may degrade the performance of floating-point programs due to the sharing of resources in that cluster.

We have measured the performance of the cost-effective clustered architecture with the general balance scheme for several SpecFP95 [97] benchmarks. Figure 3-18 shows the speed-ups of the cost-effective architecture. We can see that none of the programs is slowed down and even in some of them the speed up is significant (7% in turb3d). On average, floating point programs perform a 3.2% better. When the FP cluster has a high workload (its resources are heavily demanded by FP instructions), the balance mechanism will refrain the steering logic from sending integer instructions to that cluster, so that they do not compete for the FP resources. On the other hand, in periods of few FP calculations, the balance mechanism will send some integer instructions to the FP cluster and we could expect some speed-ups in this case.

## 3.5 Conclusions

We have proposed a number of slice-based mechanisms that dynamically partition a sequential program into the different clusters of a clustered microarchitecture. The slice-based schemes assign clusters to some groups of dependent instructions called slices. A similar concept was used in some early static code assignment scheme that attempted to exploit the potential of a cost-effective two-clustered architecture with the capability to execute simple integer instructions in both the integer and the FP datapaths [87]. Thus, in order to facilitate comparisons, our evaluations focus on a similar cost-effective microarchitecture, and we show that our dynamic schemes outperform the previous static proposal, due to the much better ability to keep cluster workloads balanced.

The different proposed schemes have different levels of performance that are explained by their effectiveness to both reduce/hide inter-cluster communications and balance the workload of clusters. We have shown that all the schemes provide a significant speedup over a conventional 8-way microarchitecture (4int + 4fp). For instance, the general balance steering achieves an average speed-up of 36% for the SpecInt95, and just an 8% below an upper bound. In other words, it enables the cost-effective architecture to realize most of its potential, with integer programs. We have also shown that this scheme significantly outperforms a previous dynamic proposal [70], because it generates a significantly lower number of inter-cluster communications.

Our results also prove that, with a cost-effective architecture, idle floating-point resources can be profitably exploited to speed-up integer programs, with minimal hardware support and no ISA change. Moreover, we show that such a cost-effective architecture also performs better than a conventional architecture when running the SpecFP95 benchmarks, even though the FP-cluster datapath is shared by integer and FP instructions.

Finally, we also show that with our steering schemes, the distributed register file has very few replicated registers. Overall, each cluster's register file has few ports and few registers, which results in a very low latency register file with very low power dissipation.