Inherently Workload-Balanced Clustered Microarchitecture

Jaume Abella*, Antonio González*+

* *Computer Architecture Department*
Universitat Politècnica de Catalunya
Barcelona (Spain)
jabella@ac.upc.edu

+ *Intel Barcelona Research Center*
Intel Labs, Universitat Politècnica de Catalunya
Barcelona (Spain)
antonio@ac.upc.edu

Abstract

The performance of clustered microarchitectures relies on steering schemes that try to find the best trade-off between workload balance and inter-cluster communication penalties. In previously proposed clustered processors, reducing communication penalties and balancing the workload are opposite targets, since improving one usually implies a detriment in the other. In this paper we propose a new clustered microarchitecture that can minimize communication penalties without compromising workload balance. The key idea is to arrange the clusters in a ring topology in such a way that results of one cluster can be forwarded to the neighbor cluster with a very short latency. In this way, minimizing communication penalties is favored when the producer of a value and its consumer are placed in adjacent clusters, which also favors workload balance.

The proposed microarchitecture is shown to outperform a state-of-the-art clustered processor. For instance, for an 8-cluster configuration and just one fully pipelined unidirectional bus, 15% speedup is achieved on average for FP programs.

1. Introduction

Clock rates have undergone a continuous increase since the first microprocessor as a result of deeper pipelines and the use of ever smaller and faster transistors. On the other hand, on-chip communications are more critical from generation to generation since they become slower in terms of number of cycles [1][10]. Wire delays are making processor designers to devote more effort and resources to techniques to minimize their impact. Clustered microarchitectures [6][7][9][12][13][14] are becoming a widely-used approach to tackle this problem.

Clustered processors deal with the wire delay problem trying to keep locally most of the communications and, at the same time, balancing the workload. Conventional clustered processors are laid out in such a way that wire delays inside a cluster are short while inter-cluster delays are long. Because of that, minimizing the penalties of wire delays and balancing the workload of the clusters are opposite objectives. The best performance is achieved when the best trade-off between these two factors is identified. Different approaches that search for this trade-off are described in the related work section.

Clustered microarchitectures are also effective at reducing energy consumption [20]. They are also effective at reducing the temperature of the chip through a better distribution of the activity across the whole die. This may translate in significant benefits in performance, by reducing the frequency of thermal emergencies, and cost, by allowing cheaper cooling solutions for a given performance level. However, conventional clustered microarchitectures tend to concentrate the activity in the minimum number of clusters that can provide the maximum throughput required by each particular code, since spreading the activity across all clusters implies an increase in communication penalties.

In this paper we propose an alternative way to implement dynamically-scheduled clustered micro-architectures. The key point is to lay out the processor in such a way that the results of a cluster can be forwarded to the neighbor cluster with a very short latency. In particular, the typical bypass network that in conventional designs allows values to be bypassed from the output of a unit to the input of any other unit of the same cluster, is replaced by a bypass network that allows values to be fast bypassed to the next cluster, in a unidirectional ring topology. In this microarchitecture, minimizing communication penalties is favored when the producer of a value and its consumer are placed in adjacent clusters, which also favors workload balance.

Note also that for codes with very small ILP, a conventional clustered processor may choose to execute most of the instructions in just one cluster, whereas the rest remain almost idle, in order to maximize performance. The proposed microarchitecture will still distribute the work evenly across all clusters, even if all have low utilization, since this also minimizes communication penalties.

The proposed microarchitecture is shown to outperform a state-of-the-art clustered processor. Average speedups of 15% are achieved for FP programs with reasonable future configurations.

The rest of this paper is organized as follows. Section 2 introduces some related work. Section 3 presents the proposed scheme. Section 4 evaluates the performance of the proposed approach and compares it with conventional clustered microarchitectures. Section 5 summarizes the main conclusions of this work.

2. Related Work

Conventional clustered architectures have fast interconnects for propagating intra-cluster signals, whereas inter-cluster communications require long and slow wires. These architectures rely on keeping most of the communications local within clusters to achieve high performance. Even if most of the communications are local, inter-cluster communications are required in a non-negligible number of cases (1 communication every 4 instructions may be common in a 4-cluster configuration [13]). Those inter-cluster communications are critical since increasing their number and their latency degrades performance significantly [2][4][5][13][14].

Some previous works on clustered microarchitectures focus on 2 clusters [6][9]. Future microprocessors are likely to have a higher degree of clustering. These architectures rely on a mechanism to distribute instructions across clusters that is referred to as steering logic [15]. Some approaches are based on partitioning the code at branch boundaries. Trace processors [7][16][18][19] partition dynamically the code into chunks of consecutive instructions called traces. Then, each trace is steered as a unit to a given cluster. Since the traces have similar length, the workload is effectively balanced. However, this scheme may incur in a large number of communications.

Multiscalar processors [7][18] divide the code into tasks. Each task is made up of a set of consecutive instructions and is assigned to a different processing element, which are interconnected through a ring network. PEs have fast internal interconnects and slow connections to other PEs. Thus, communication among tasks use the slow inter-cluster connection. This is clearly different to our approach, which has fast connections to the following cluster in the ring, no interconnects inside each cluster, and slow connections to other clusters. Besides, our approach distribute the code to clusters at run time, through a per-instructions scheme, whereas Multiscalar steers tasks that are determined at compile time.

Other works are based on steering instructions considering data dependences, trying to send dependent instructions to the same cluster without compromising workload balance [4][5][6][11][12][13][14][17]. The Multicluster architecture [6] partitions the register name space into two subsets. The program is partitioned at compile time by estimating the workload balance and inter-cluster communication.

Palacharla *et al.* [12] propose a dependence-based clustered architecture where each issue queue is a FIFO queue. This scheme places in the same FIFO only instructions that must be executed sequentially.

Some other mechanisms [4][5][13][14] have been proposed to deal with the problem of reducing the number of required communications and maximizing workload balance at the same time. These policies are based on using a dependence-based steering algorithm and an additional mechanism to manage workload balance. A recent mechanism [14] is deeply described in the evaluation section since it is used for comparison purposes.

3. Ring Clustered Processor

This section describes the proposed processor organization, which will be referred to as *ring clustered* microarchitecture. Whereas conventional clustered processors have fast interconnects between the outputs and the inputs of the functional units inside the same cluster, our approach is based on having these bypasses between the outputs of the functional units of a given cluster and the inputs of the functional units of the *following* cluster. The clusters are arranged forming a ring in such a way that cluster 0 bypasses its data to cluster 1, cluster 1 to cluster 2 and so on. Finally, cluster *n*-1 bypasses its data to cluster 0, closing the ring. We also assume such fast inter-cluster bypasses for tags in order to perform the wakeup in the following cluster of the ring, instead of waking up instructions in the same cluster. Additionally, there is a set of buses communicating values from one cluster to a cluster other than the following one. They are unidirectional and fully pipelined buses. This kind of buses can be easily designed, have low latency per hop in comparison with non-pipelined buses and scale quite well.

Figure 1 shows a block diagram of the ring clustered microarchitecture. It can be seen that data produced in one cluster are sent to the next cluster and there are not bypasses from the outputs of the functional units of a given cluster to the input of these functional units. The register file is distributed across all the clusters. Each register file can be read only from the cluster where it is and written from the previous cluster in the ring. This organization allows the processor to issue dependent instructions back to back only if they are sent to contiguous clusters. Instructions issued in a given cluster wakeup instructions just in the following cluster of the ring, but not in the same cluster. Section 3.2 shows that these assumptions are realistic.

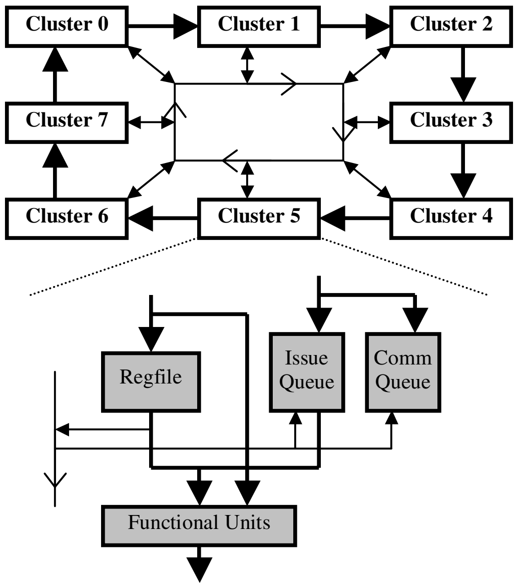

Figure 1. Ring clustered microarchitecture

This architecture works as follows. Assume an instruction is issued in cluster *i*. When it finishes its execution, the output is written in the register file of the following cluster (cluster (*i*+1) mod *N*, where *N* stands for the number of clusters). This datum is also bypassed to the functional units of the following cluster. Instructions in the issue queue and communication instructions of the following cluster are woken up. Communication instructions are generated dynamically when an operand is needed in a cluster where it is not present and wait in a separate issue queue. When an instruction is dispatched, for each required communication, a new communication instruction is created in the producer cluster (the one where the value is stored), and one register is allocated in the consumer cluster for storing the copy of the corresponding value. More details can be found elsewhere [13][14].

We assume homogeneous clusters, all with the same configuration although the proposed scheme can be applied to heterogeneous clusters.

The data cache is centralized, forming a cluster. It has been assumed a delay to send the addresses and the same delay to send the read datum back to the cluster that requested it, in addition to the latency of the cache.

The register file has enough ports to perform all the required accesses in a given cycle. Thus, if each cluster has issue width *IW*, *NumFU* functional units and there are *B* buses, the number of ports is:

$$\text{Read Ports} = IW \times 2 + B$$
$$\text{Write Ports} = NumFU + B$$

This configuration guarantees that any issued instruction can read up to two operands and, if the buses are idle, a communication instruction per bus can also be issued. On the other hand, the register file has write ports for the data produced by the functional units of the previous cluster and the incoming buses. A lower number of register ports may well suffice to provide the same performance but this analysis is out of the scope of this paper.

A ring clustered processor, like conventional clustered processors, copies data from one cluster to another only when needed. A copy flows through a bus until it reaches the destination register file. Multiple copies of a given register can be present in different clusters and all copies are released at the same time [13][14]. Alternatively, register copies could be released as soon as they are read, whereas the original copy is released when the instruction that redefines the register commits. This would reduce register pressure at the expense of an increase in the number of copies. In this paper we just analyze the former alternative.

3.1 Steering Algorithm

A simple dependence-based steering policy is used since, in addition to reducing communication, it also balances the workload. The algorithm works as follows:

```
If the instruction has 0 source operands:
    The cluster with more free registers is
    chosen.
If the instruction has 1 source operand:
    Those clusters where the register is mapped
    are selected, and the one with more free
    registers among them is chosen.
If the instruction has 2 source operands:
    If there is at least one cluster where both
    operands are mapped:
        Those clusters where both registers are
        mapped are selected, and the one with
        more free registers among them is chosen.
    Else:
        Those clusters where one operand is
        mapped are chosen. Since one
        communication is required, it is chosen
        the one that incurs in the shorter
        communication distance. If there is more
        than one, the one with more free
        registers among them is chosen.
If the chosen cluster is full, then the dispatch
stage is stalled.
```

The algorithm sends the instructions to the clusters considering their dependences. In case of having more than one candidate cluster to dispatch an instruction, the one with more registers available is selected.

The following example illustrates the above algorithm. Figure 2 shows the source code as well as the steering decisions taken for each instruction. It can be observed that, when an instruction is steered to a given cluster, the value is only available in the following cluster of the ring.

```
I1. R1 = 1
I2. R2 = R1 + 1
I3. R3 = R1 + R2
I4. R4 = R1 + R3
I5. R5 = R1 x 3
```

Registers for each cluster:

0	1	2	3

I1 is sent randomly to 0

	R1		

I2 is sent to 1 (R1 is local)

	R1	R2	

I3 is sent to 2 (R2 is local, R1 requires only 1 cycle of bus)

	R1	R1,R2	R3

I4 is sent to 3 (R3 is local, R1 requires only 1 cycle of bus from 2)

R4	R1	R1,R2	R1,R3

I5 can be sent to 1, 2 or 3. Cluster 3 has more free registers

R4,R5	R1	R1,R2	R1,R3

Figure 2. Example of the steering algorithm

Whereas a conventional dependence-based clustered architecture partitions vertically the dependence graph, sending dependent instructions to the same cluster if the workload imbalance is not so high, the ring clustered processor partitions the dependence graph in a horizontal-like approach. Another interesting feature is that those instructions with two operands are always sent to a cluster where at least one of its operands is mapped. Thus, an instruction never requires two communications.

3.2 Layout Considerations

As it has been outlined in previous subsections, our approach relies on having fast interconnects between one cluster and the following one instead of having fast intra-cluster wires. The purpose of this subsection is to verify that this assumption is realistic. In order to do so, it is necessary to do a high-level layout to check that the distance of neighbor connections are short enough to bypass data with the same or shorter delay than they would have for the intra-cluster connections of a conventional clustered microarchitecture.

The cluster placement for an 8-cluster configuration is shown in Figure 3. It can be observed that two different modules are required for an 8-cluster configuration: straight and corner clusters. For a 4-cluster configuration only corner clusters are required.

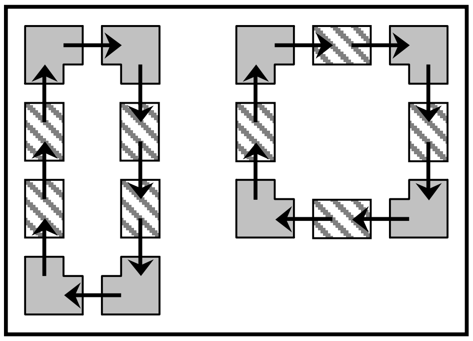

Figure 3. Placement alternatives for 8 clusters

Designing circuits at high level to deduce the layout is hard to approximate, since it is strongly dependent on the technology and the circuits characteristics. Designing the whole backend at circuit level and doing a full layout is the only way to get exact figures for the area and delays of the different components. Such effort is unaffordable to us and probably unreasonable for a microarchitectural study. Our objective is to convince of the feasibility of the proposed microarchitecture, and for that we use alternative schemes based on some models.

Based on published models [8] we have estimated the area of the different components of a given cluster. We have used the same parameters than the authors of the model. They can be found in Table 1. For the sake of simplicity we have assumed that all components but the queues are square blocks.

Other functional units not detailed in the table can be assumed to be out of the critical path even if this fact increases their latency since they do not execute frequent instructions. The area of a register file cell is based on what the model suggests as average area after looking at several current microprocessors. This assumption may well be pessimistic for our clusters. For instance, the model [8] reports that a register file with 4 read and 2 write ports has a cell area of 27200 λ^2. If we consider that each cluster is able to issue 1 INT + 1 FP instructions per cycle and there is one global bus, 3 read and 3 write ports per register file (integer or FP one) are enough. Thus, with the same number of ports (3R+3W instead of 4R+2W) we have assumed larger register file cells (40600 λ^2 instead of 27200 λ^2).

Table 1. Area of the main cluster's blocks

Component	Area per cell (λ^2)	Size	Height/Width & Total area (λ/λ & λ^2)
Issue queue	22.300 CAM +13.900 RAM	16 entries, 12 bits CAM/entry, 24 bits RAM/entry	9.619 / 1.000 9.619.200
Comm. queue	22.300 CAM +13.900 RAM	16 entries, 6 bits CAM/entry, 9 bits RAM/entry	8.006 / 1.000 8.006.400
Register file	40.600	48 regs, 64 bits/reg	11.168 / 11.168 124.723.200
Integer ALU	2.410.000	64 bits	12.419 / 12.419 154.240.000
Integer Multiplier	1.840.000	64 bits	10.852 / 10.852 117.760.000
FP Unit (Add+Mult)	4.550.000	64 bits	17.065 / 17.065 291.200.000

The next step consists on placing the different components of a cluster in such a way that they can be easily connected (inputs of one cluster are close to outputs of the previous cluster), and the wires from one cluster to the following one have similar length to that of an intra-cluster communication in a conventional clustered microarchitecture. Since the largest block is the FP unit, and its height (or width) is around 17.100 λ, the design for both types of cluster modules (straight and corner ones) would require intra-cluster connections of this order of magnitude.

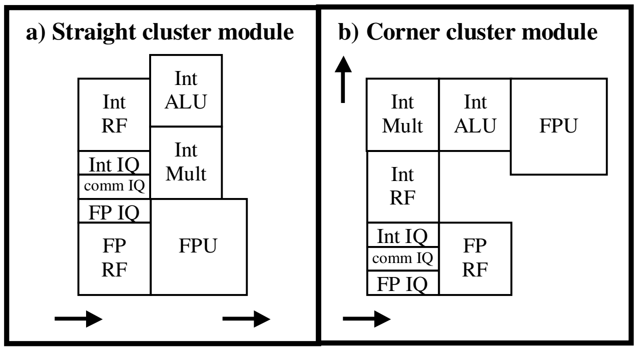

Figure 4. High level layout for cluster modules

Figure 4 shows the proposed design for both types of modules. It can be observed that the maximum length between the input and the output of two cluster modules for integer data is 17.400 λ (17.100 – 10.900 + 11.200) from the output of the integer multiplier of a straight module to the input of any integer functional unit of another straight module. For FP data, the maximum distance exists when any module is connected to a corner one: 23.300 λ (12.400 + 10.900). Thus, only FP data may have their bypass delay increased. However, if the FP unit could fill the empty space in the middle of the corner module, the delay for the worst case could be reduced to 10.900 λ.

To conclude, it seems feasible to send data from one cluster to the following one with similar delay to that of an intra-cluster bypass of a conventional clustered processor. Thus, we assume that a given instruction can be issued back to back with its predecessor, which is in the previous cluster.

If shorter inter-cluster connections are desired, the ring-like approach can be designed with two independent rings: one for integer instructions and another for FP ones. Thus, integer and FP clusters can be smaller and the blocks can be placed in such a way that inter-cluster bypasses are shorter. Instructions with integer inputs and FP output or vice versa are extremely rare, so it can be assumed a bi-directional bus from one of the integer clusters to one of the FP clusters for those instructions. The only frequent instructions with integer inputs and FP outputs are FP load instructions. The address calculation of these instructions is sent to the integer ring, and when it has been computed, it is sent to the *LSQ*. Once the memory access is performed, the datum is sent back to the corresponding cluster of the FP ring. Thus, these instructions work in the same way even if FP and integer units are in different rings.

If a further reduction in bypass delay is needed, fatter connections and/or repeaters [10] can be used.

The high level layout for the clusters of the architecture with two independent rings (one for integer instructions and another for FP ones) is shown in Figure 5. It can be observed that the maximum distance for integer or FP data is 11.200 λ, which correspond to any module connected to a straight one.

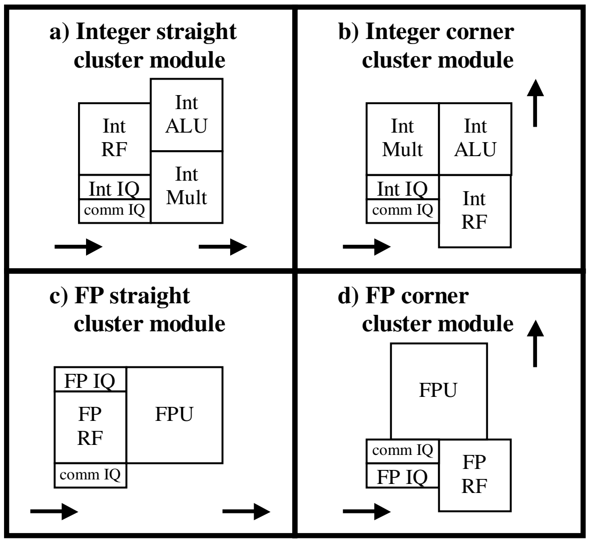
a) Integer straight cluster module
Int RF
Int ALU
Int Mult
Int IQ
comm IQ
b) Integer corner cluster module
Int Mult
Int ALU
Int IQ
comm IQ
Int RF
c) FP straight cluster module
FP IQ
FP RF
FPU
comm IQ
d) FP corner cluster module
FPU
comm IQ
FP IQ
FP RF

Figure 5. High level layout for cluster modules with integer and FP independent rings

3.3 Additional Comments

Some considerations must be taken into account for the design of the ring clustered processor. It is hard to achieve accurate floorplans of the clusters without doing the full detailed layout of the whole processor core, but we believe that this is a first-order approximation that validates the potential of the idea. The objective is to show some evidence that issue queues, register files and functional units can be laid out in such a way that sending the tags/data from one cluster to the following one in the ring has a similar delay to that of the intra-cluster connections of a conventional clustered architecture.

In this work, the distance in time to/from the data cache and the Load/Store queue (LSQ) has been considered to be the same for all clusters. We have assumed a 1-cycle penalty for sending data to/from these structures to any cluster. In some implementations the cache latency may not be uniform across all the clusters. That would probably degrade performance, but it is expected that the effect will be the same for both a ring and a conventional clustered architectures. Note also that the cache could be partitioned in a clustered architecture so that each cluster had a local cache that could be accessed very fast. However, this is orthogonal to the main ideas proposed in this work.

4. Evaluation

The proposed architecture is evaluated in this section. First, we describe the conventional clustered microarchitecture used for comparison purposes and the experimental framework that has been used. Results are then reported.

4.1 Microarchitecture Used for Comparison

The proposed ring clustered microarchitecture will be compared with a state-of-the-art clustered microarchitecture [14] with about the same number of resources: number and configuration of clusters and buses, number of bypasses, etc. In the rest of the paper, the architecture used for comparison purposes will be referred to as *Conv*, whereas the ring clustered microarchitecture will be referred to as *Ring*.

The *Conv* processor has clusters with fast intra-cluster connections and point-to-point buses to communicate data to remote clusters. This processor tries to steer instructions to the cluster where their source operands are mapped, but in case of requiring communications, the steering algorithm chooses the cluster that minimizes the communication delay. This algorithm requires workload imbalance control. The figure used to measure this feature is DCOUNT. The interested reader is referred to the original paper for details[14].

The detailed steering algorithm is as follows.

```
If the workload imbalance is higher than the
threshold:
    The least loaded cluster is chosen (that with
    lower DCOUNT value).
Else:
    If any source operand is not available at
    dispatch time:
        Cluster(s) where the pending operand(s)
        are to be produced are selected
    If all source operands are available at
    dispatch time:
        Cluster(s) that minimize the longest
        communication distance are selected.
    If it has no source operands:
        All clusters are selected.
    The least loaded cluster among the selected
    clusters is chosen.
```

It can be observed that the algorithm tries to reduce the number of inter-cluster communications and balance the workload at a time. The latency to steer instructions to clusters is 1 cycle for both *Conv* and *Ring* architectures. Larger latencies may be considered but they will have a minor impact on the conclusions since each additional cycle in the frontend will basically increase the penalty for branch misspredictions by about one cycle in both the proposed and the baseline microarchitecture.

4.2 Experimental Framework

Performance results have been obtained through an enhanced version of Simplescalar [3]. The main enhancements are the separation of the reorder buffer and the issue queue, the extension to model register files and the split of the pipeline for integer and FP instructions. Additionally, some structures have been distributed to model clustered architectures: issue queues, register files and functional units. Three different configurations have been evaluated: 4 and 8 clusters of issue width 2 INT + 2 FP each, and 8 clusters of issue width 1 INT + 1 FP each. Table 2 describes the assumed processor configuration.

The 4-cluster configuration does not require high number of communications so it has been assumed one unidirectional fully pipelined bus (i.e. a datum can be transmitted from every cluster to the following one at the same time). The 8-cluster configuration may require more and farther communications, so either one or two buses have been considered. For the two-buses configuration, *Ring* has both buses with the same direction, whereas *Conv* has one bus for each direction in order to reduce the distance of the communications. Table 3 details how the different configurations are referred to in the rest of the paper.

Table 2. Processor configuration

Fetch, decode, commit width: *8 instructions*
Branch pred.: *Hybrid 2K Gshare, 2K bimodal, 1K selector* **BTB**: *2048 entries, 4-way*
L1 Icache: *64KB, 2-way, 32 byte line (1 cycle)* **L1 Dcache**: *32KB, 4-way, 32 byte line, 4 R/W ports (2 cycles)* **L2 unified cache**: *512KB, 4-way, 64 byte line (10 cycles hit, 100 cycles miss, 2 cycles interchunk)* **Latency to/from L1 Dcache**: *1 cycle*
Fetch queue: *64 entries* **Issue queue (4 clusters)**: *32 INT + 32 FP + 16 comm entries/cluster* **Issue queue (8 clusters)**: *16 INT + 16 FP +16 comm entries/cluster* **Reorder buffer**: *256 entries* **Load/store queue**: *128 entries*
Register file (4 clusters): *64 INT + 64 FP registers per cluster* **Register file (8 clusters)**: *48 INT + 48 FP registers per cluster*
INT functional units: *ALU (1 cycle), mult/div (3 cycles mult, 20 cycles non-pipelined div)* **FP functional units**: *ALU (2 cycles), mult/div (4 cycles mult, 12 cycles non-pipelined div)* **1 INT + 1 FP issue width**: *1 unit of each type per cluster* **2 INT + 2 FP issue width**: *2 units of each type per cluster*

Table 3. Evaluated configurations

Architect.	Num. clust.	Issue width	Num. buses	Name
Conv	4	2 INT + 2 FP	1	*Conv_4clus_1bus_2IW*
Conv	8	1 INT + 1 FP	1	*Conv_8clus_1bus_1IW*
Conv	8	1 INT + 1 FP	2	*Conv_8clus_2bus_1IW*
Conv	8	2 INT + 2 FP	1	*Conv_8clus_1bus_2IW*
Conv	8	2 INT + 2 FP	2	*Conv_8clus_2bus_2IW*
Ring	4	2 INT + 2 FP	1	*Ring_4clus_1bus_2IW*
Ring	8	1 INT + 1 FP	1	*Ring_8clus_1bus_1IW*
Ring	8	1 INT + 1 FP	2	*Ring_8clus_2bus_1IW*
Ring	8	2 INT + 2 FP	1	*Ring_8clus_1bus_2IW*
Ring	8	2 INT + 2 FP	2	*Ring_8clus_2bus_2IW*

For this study we have used the whole Spec2000 benchmark suite [21] with the *ref* input data set. This suite consists of 12 integer and 14 FP programs. We have simulated 100 million of instructions for each program after skipping the initialization part. The programs were compiled with the HP/Alpha compiler with –O4 –non_shared flags.

4.3 Performance

Figure 6 shows the speedup of *Ring* over *Conv* for each configuration. *Ring* achieves higher performance than *Conv* for all configurations. It can be observed that the speedup for integer programs is smaller than for FP programs, and even slightly negative for one configuration. Since *Ring* is much more effective than *Conv* at reducing the number and the distance of the communications, *Ring* achieves higher speedups for programs with larger number of communications, as it is the case of FP programs (see details below). This becomes even more obvious under the presence of just one bus. In this case, the speedup increases significantly. In order to show why *Ring* performs better than *Conv*, we have analyzed the penalty introduced by communications and workload imbalance.

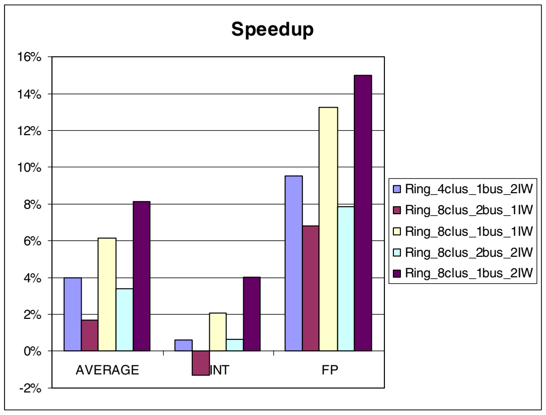

Figure 6. Speedup of *Ring* over *Conv*

4.4 Communications

Figure 7 shows the number of communications per instruction. *Ring* requires fewer communications than *Conv* because it succeeds at distributing the workload without introducing extra communications. On the other hand, *Conv* reaches quite often situations where it has to send an instruction to the least loaded cluster, even if that decision introduces more communications. Thus, the workload is balanced at the expense of more communications. It can also be observed that FP programs require more communications than integer ones.

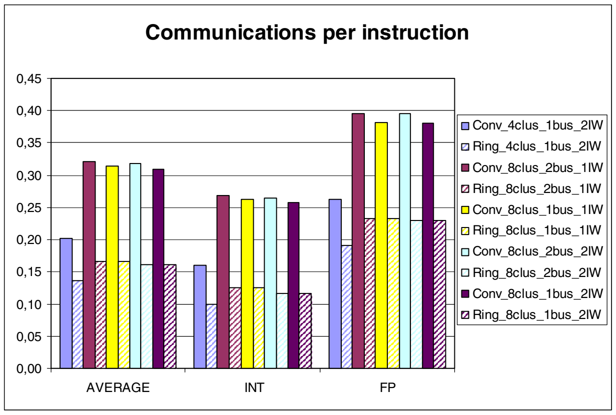

Figure 7. Average number of communications per instruction

Communications distance has to be also studied. The distance of a communication is the number of hops required to copy the data from the source cluster to the destination cluster. Shorter communications are desirable in order to reduce the time that consumer instructions spend waiting for the remote data. Figure 8 shows the average distance per communication for the different configurations. It can be observed that *Conv* has similar delay to *Ring* for two buses, but *Ring* has much shorter communications for one bus.

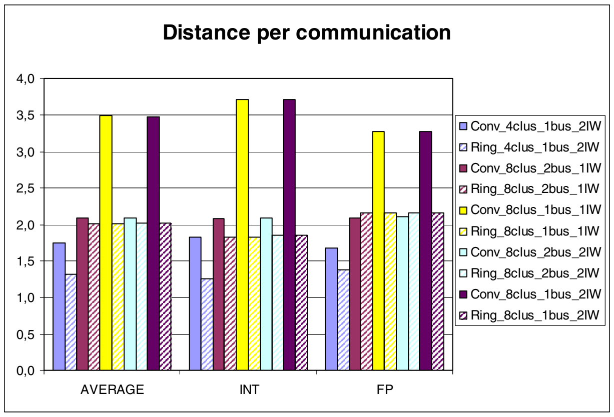

Figure 8. Average distance per communication

Both, the number of communications and their distance, determine the bus occupancy and thus, further delays due to bus contention. Figure 9 shows the average number of cycles that a ready communication instruction has to wait until it can access the bus. It can be seen that *Conv* has much higher contention than *Ring*, especially if there is only one bus. For both 8-cluster, 1-bus configurations *Conv*'s contention is larger than 5 cycles for FP instructions.

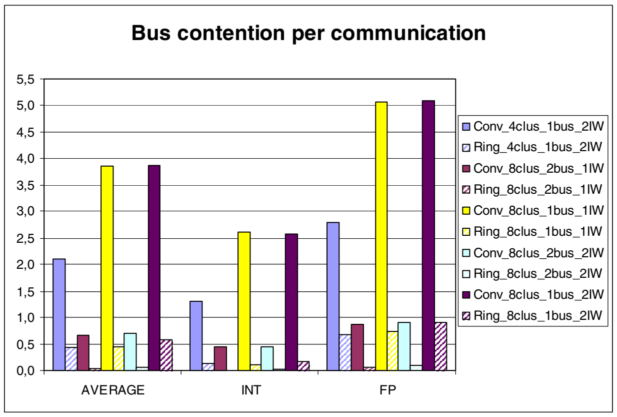

Figure 9. Average delay per communication due to bus contention

4.5 Workload Imbalance

The workload balance figure used to guide the steering algorithm of *Conv* is DCOUNT since it provides better performance than others, but in order to show the effect of workload imbalance in IPC it has been used another more suitable figure: NREADY[13][14]. NREADY accounts for the number of ready instructions that are not issued at a given instant of time due to exceeding the issue width of their respective clusters, but could be issued in other clusters since they have idle functional units.

Figure 10 quantifies the workload imbalance for different configurations. It can be observed that the conventional clustered microarchitecture balances the workload somewhat better than the ring clustered processor. However, this small detriment in workload balance is more than offset by the reduction in communication penalties, as shown above, especially for FP programs, which are more communication intensive. *Conv* reduces the workload imbalance at the expense of a much larger number of communications, as shown above, which produces significant performance degradation.

The configurations with 8 clusters and 2 INT + 2 FP issue width are especially interesting since they achieve a very good workload balance for both *Conv* and *Ring*, and are still suitable for high clock rates since the structures involved in the issue process are quite small: 16-entry issue queues and 48-entry register files.

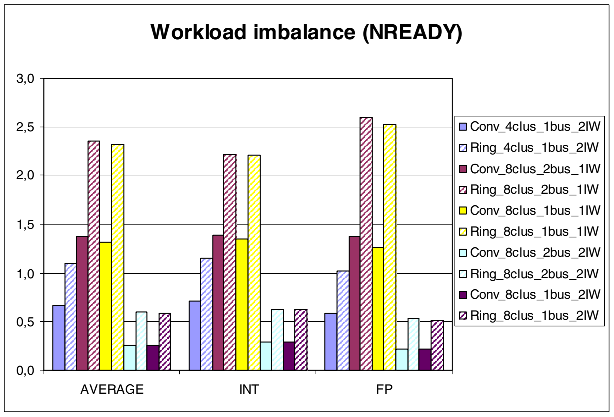

Figure 10. Workload imbalance using NREADY figure

Conv uses DCOUNT figure for balancing workload, so it is expected that the number of instructions dispatched to each cluster is approximately the same. On the other hand, *Ring* does not use any mechanism for balancing workload since it is inherent to the dependence-based steering algorithm. Nevertheless, it is interesting to show how many instructions are dispatched to each cluster. Figure 11 shows the percentage of instructions dispatched to each cluster for all benchmarks for the *Ring_8clus_1bus_2IW* configuration. It can be observed that the percentage of instructions dispatched to each cluster is pretty much the same for all programs. Similar results are achieved for the other configurations.

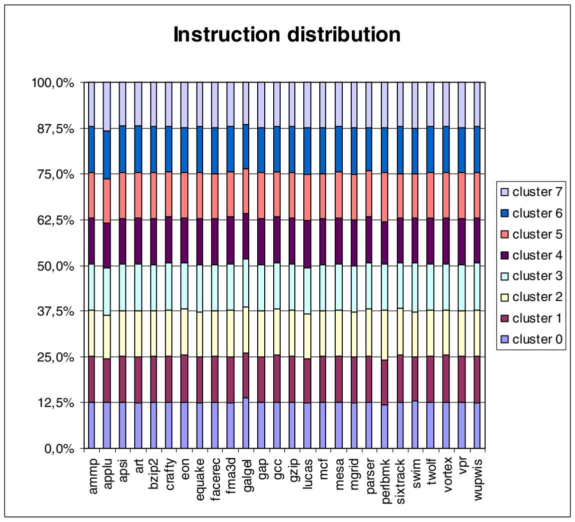

Figure 11. Distribution of the dispatched instructions across the clusters

4.6 Scaling Wires

It is well known that wires scale very badly, so it is expected that future clustered microprocessors may have large latencies for inter-cluster communications. We have assumed buses with 1-cycle latency per hop, but this may not be feasible for future processors. Thus, it is interesting to analyze how both, the *Conv* and the *Ring* processors, perform with slower buses. For this purpose, we have evaluated the configurations with 8 clusters and 2 INT + 2 FP issue width (with 1 and 2 buses) using 2-cycle latency per hop, and fully pipelined buses. Thus, a given bus may be processing 16 communications at a time.

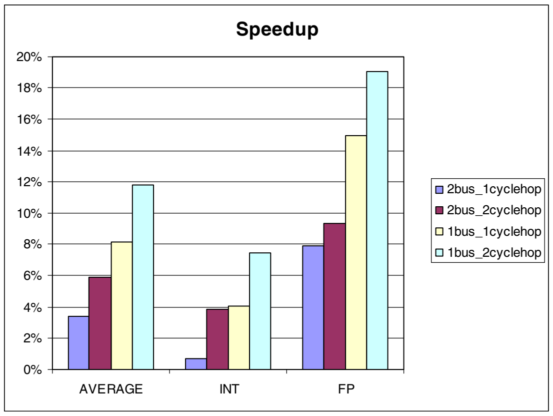

Figure 12. Speedup of *Ring* over *Conv* for different bus latencies

Figure 12 shows the speedup of *Ring* over *Conv*. For the configuration with one bus the speedup grows from 8.1% with 1-cycle latency per hop to 11.8% (19.1% for FP programs) with 2-cycle latency per hop. Similar trend is observed for 2 buses. *Conv* loses much more performance than *Ring* because the former has more and longer communications than the latter.

4.7 A Simple Steering Algorithms

The steering algorithm that we have assumed has similar complexity to the one by Parcerisa et al [14]. In this section we evaluate the performance of a conventional clustered microarchitecture and the ring one using a simpler steering algorithm, which complexity is similar to that of the rename logic. The algorithm (SSA) is as follows:

```
If the instruction has at least one input
operand:
    It is send to the lower index cluster that
    stores (or will store) its leftmost operand.
Else (no input operands):
    It is sent to a cluster in a round-robin
    fashion.
```

As it can be seen, this simple steering algorithm does not include an explicit workload imbalance control.

Figure 13 shows the speedup of *Ring* over *Conv* when the simple steering algorithm is used. It can be observed that the speedup is huge. For instance, for a 8-clusters 1-way issue, 2-bus configuration, the speedup of *Ring* over *Conv* is 50% in average (1.5X). As observed for the other steering, the speedup for FP programs (80%) is higher than for integer programs (30%).

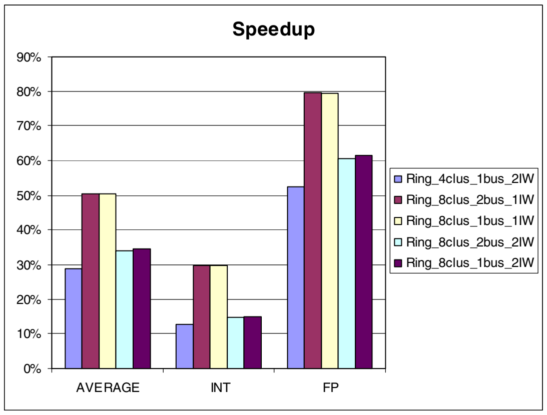

Figure 13. Speedup of Ring+SSA over Conv+SSA

The performance drop of *Ring+SSA* with respect to *Ring* is between 5% and 14% depending on the configuration. The performance drop is small because the workload balance is similar whereas the communication distance slightly increases. The performance drop of *Conv+SSA* with respect to *Conv* is between 23% and 42% depending on the configuration. It is so high mainly due to the workload imbalance. *Ring* inherently balances the workload, whereas *Conv* does not. Thus, *Conv+SSA* tends to concentrate most of the instructions in very few clusters. This concentration reduces the communication penalty but incurs in many dispatch stalls because the cluster selected to steer instructions is full. Additionally, full clusters hold many ready instructions that cannot be issued because there are more of them than the issue width, whereas other clusters have less workload that they could absorb. In fact, due to this workload imbalance (Figure 14), *Ring+SSA* shows higher speedup with respect to *Conv+SSA* when the issue width is 1. On the other hand, for the enhanced steering (see Figure 6) we observed the opposite trend: lower speedup when the issue width is 1.

Note that the workload imbalance of *Ring+SSA* is 10% higher than that of *Ring* (see Figure 10). In the case of *Conv*, the workload imbalance increases by between 100% and 300% depending on the configuration.

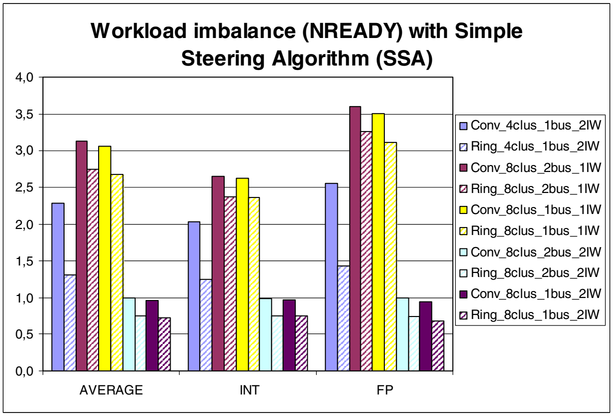

Figure 14. Workload imbalance using NREADY figure with Simple Steering Algorithm

5. Conclusions

In this paper we have presented a new clustered microarchitecture for superscalar processors. A distinguishing feature of this microarchitecture is that those schemes that favor hiding wire delays also favor workload balance among the clusters, due to the particular way that clusters are interconnect. The clusters are arranged in a ring topology, which is not new, but unlike previous proposals, fast interconnects are used for forwarding values among neighbor clusters, whereas internal bypasses are not needed. As a consequence, a dependence-based steering algorithm that attempts to reduce the number and distance of global communications is extremely effective at distributing the workload across all the clusters without requiring any explicity workload balance scheme.

The proposed ring clustered microarchitecture significantly outperforms state-of-the-art clustered organizations. The benefits increase as the number of clusters increases, and the global interconnects are simple, scarce and have long latencies. Thus, the ring clustered microarchitecture is more scalable than conventional ones. For instance, for an 8-cluster configuration and just one fully pipelined unidirectional bus with a latency of 2 cycles per hop, the proposed architecture achieves 19% speedup over a state-of-the-art, topology-aware conventional clustered architecture for FP programs. If a simpler steering algorithm is used, the speedup can be as large as 50% on average for the whole Spec2000 benchmark suite.

While conventional clustered architectures tend to steer the instructions to one or a few number of clusters until a certain workload imbalance is achieved, the proposed ring-like architecture distributes the activity across all the clusters during all the time. This fact results in a better temperature distribution across the chip and thus, it is expected to reduce the frequency of temperature emergencies with respect to a conventional clustered microarchitecture.

To conclude, the proposed clustered microarchitecture outperforms state-of-the-art microarchitectures. Besides, it is more scalable than conventional ones, requires less communication resources and is more effective at distributing the activity across all the clusters.

Acknowledgements

This work has been partially supported by the Ministry of Education and Science under grants AP2002-3677, TIN2004-07739-C02-01 and TIN2004-03072, the CICYT project TIC2001-0995-C02-01, Feder funds, and Intel Corporation. We would like to thank the anonymous reviewers by their comments.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, D. Burger. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures". In proceedings of the 27th International Symposium on Computer Architecture (ISCA'00), June 2000.

[2] A. Baniasadi, A. Moshovos. "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors". In proceedings of the 33th International Symposium on Microarchitecture (MICRO'00), December 2000.

[3] D. Burger, T. Austin. "The Simplescalar Tool Set, Version 3.0". Technical Report, Computer Sciences Department, University of Wisconsin-Madison, 1999.

[4] R. Canal, J.M. Parcerisa, A. González. "A Cost-Effective Clustered Architecture". In proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99), October 1999.

[5] R. Canal, J.M. Parcerisa, A. González. "Dynamic Cluster Assignment Mechanisms". In proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA'00), January 2000.

[6] K.I. Farkas, P. Chow, N.P. Jouppi, Z. Vranesic. "The Multicluster Architecture: Reducing Cycle Time through Partitioning". In proceedings of the 30th International Symposium on Microarchitecture (MICRO'97), December 1997.

[7] M. Franklin. "The Multiscalar Architecture". Ph.D. thesis, C.S. Department, University of Wisconsin-Madison, 1993.

[8] S. Gupta, S.W. Keckler, D. Burger. "Technology Independent Area and Delay Estimates for Microprocessor Building Blocks". Technical Report 2000-5, Department of Computer Sciences, University of Texas at Austin, April 2000.

[9] L. Gwennap. "Digital 21264 Sets New Standard". Microprocessor Report, 10 (14), October 1996.

[10] R. Ho, K.W. Mai, M.A. Horowitz. "The Future of Wires". In proceedings of the IEEE, 89(4), 490-504. April, 2001.

[11] R. Nagarajan, K. Sankaralingam, D. Burger, S.W. Keckler. "A Design Space Evaluation of Grid Processor Architectures". In proceedings of the 34th International Symposium on Microarchitecture (MICRO'01), December 2001.

[12] S. Palacharla, N.P. Jouppi, J.E. Smith. "Complexity-Effective Superscalar Processors". In proceedings of the 24th International Symposium on Computer Architecture (ISCA'97), June 1997.

[13] J.M. Parcerisa, A. González. "Reducing Wire Delay Penalty through Value Prediction". In proceedings of the 33th International Symposium on Microarchitecture (MICRO'00), December 2000.

[14] J.M. Parcerisa, J. Sahuquillo, A. González, J. Duato. "Efficient Interconnects for Clustered Microarchitectures". In proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02), September 2002.

[15] N. Ranganathan, M. Franklin. "An Empirical Study of Decentralized ILP Execution Models". In proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98), October 1998.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, J.E. Smith. "Trace Processors". In proceedings of the 30th International Symposium on Microarchitecture (MICRO'97), December 1997.

[17] A. Seznec, E. Toullec, O. Rochecouste. "Register Write Specialization Register Read Specialization: a Path to Complexity-Effective Wide-Issue Superscalar Processors". In proceedings of the 35th International Symposium on Microarchitecture (MICRO'02), November 2002.

[18] G. Sohi, S. Breach, T.N. Vijaykumar. "Multiscalar Processors". In proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95), June 1995

[19] S. Vajapeyam, T. Mitra. "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences". In proceedings of the 24th International Symposium on Computer Architecture (ISCA'97), June 1997.

[20] V. Zyuban. "Inherently Lower Power High Performance Superscalar Architectures". Ph. D. thesis, University of Notre Dame, January 2000.

[21] SPEC2000. http://www.specbench.org/osg/cpu2000/