A Cost-Effective Clustered Architecture

Ramon Canal, Joan-Manuel Parcerisa, Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Cr. Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain
{rcanal,jmanel,antonio}@ac.upc.es

Abstract

In current superscalar processors, all floating-point resources are idle during the execution of integer programs. As previous works show, this problem can be alleviated if the floating-point cluster is extended to execute simple integer instructions. With minor hardware modifications to a conventional superscalarprocessor, the issue width can potentially be doubled without increasing the hardware complexity. In fact, the result is a clustered architecture with two heterogeneous clusters.

In this paper, we propose to extend this architecture with a dynamic steering logic that sends the instructions to either cluster. The performance of clustered architectures depends on the inter-cluster communication overhead and the workload balance. We present a scheme that uses run-time information to optimise the trade-off between these figures. The evaluation shows that this scheme can achieve an average speed-up of 35% over a conventional 8-way issue (4 int + 4 fp) machine and that it outperforms the previous proposed one.

1. Introduction

It is well known that current superscalar organisations are approaching a point of dimishing returns. It is not trivial to change from a 4-way issue to an 8-way issue architecture due to its hardware complexity and its implications in the cycle time. Nevertheless, the instruction level parallelism (ILP) that an 8-way issue processor can exploit is much beyond that of a 4-way issue one. One of the solutions to this challenge is clustering. Clustering offers the advantages of the partitioned schemes where one can achieve high rates of ILP and sustain a high clock rate. A partitioned architecture tends to make the hardware simpler and its control and data paths faster. For instance, it has fewer register file ports, fewer data bus sources/destinations, and fewer alternatives for many control decisions.

Current processors are partitioned into two subsystems (the integer and the floating point one). As it has been shown [14, 16], the FP subsystem can be easily extended to execute simple integer and logical operations. For instance, both register files nowadays hold 64-bit values, and simple integer units (no multiplication and division) can be embedded within a small hardware cost due to today's transistor budgets. Furthermore, the hardware modifications required of the existing architectures are minimal. This work focuses on this type of clustered architecture with two clusters, one for integer calculations and another one for integer and floating-point calculations. The advantage of this architecture is that now its floating-point registers, data path and mainly, its issue logic are used 100% of the time in any application.

There are two main issues concerning clustered architectures. The first one is the communication overhead between clusters. Since inter-cluster communications can easily take one or more cycles, the higher the number of communications the lower the performance will be due to the delay introduced between dependent instructions. The second issue is the workload balance. If the workload is not optimally balanced, one of the clusters might have more work than it can manage and the other might be less productive than it can be. Thus, in order to achieve the highest performance we have to balance the workload optimally and, at the same time, minimise the number of communications. The workload balance and the communication overhead depend on the technique used to distribute the program instructions between both clusters. Programs can be partitioned either at compile-time (statically) or at run-time (dynamically). The latter approach relies on a steering logic that decides in which cluster each decoded instruction will be executed. The steering logic is responsible for maximising the trade-off between communication and workload balance and therefore, it is a key issue in the design. In this work, a new steering scheme is proposed and its performance is evaluated. We show that the proposed scheme outperforms a previously proposed static approach for the same architecture [16]. Moreover, compared to a conventional architecture, it achieves an average speed-up of 35% for the SpecInt95 and two Mediabench programs.

The rest of the paper is organised as follows. Section 2 presents the architecture and several approaches for the steering logic. Section 3 describes the experimental framework, the evaluation methodology, and it presents the performance improvements resulting from the architecture and steering logic described in Section 2. In Section 4 some related work is presented and we conclude in Section 5.

2. Clustered architecture

Clustered architectures have shown to be very effective to reach high issue rates with low hardware complexity and therefore they can take advantage of the ILP of the applications [2, 8, 13, 14]. Although integer programs seem not to have much parallelism, there is a growing number of integer applications with high ILP such as multimedia workloads (i.e. real-time video and audio). At the same time, cluster organisations can operate at a higher clock rate than the superscalar designs with an equivalent issue-width. In this work, we present a cost-effective clustered architecture that requires minimal modifications of a conventional superscalar organisation. This architecture is based on the proposal made by Sastry et al. [16], which is extended with dynamic steering of instructions

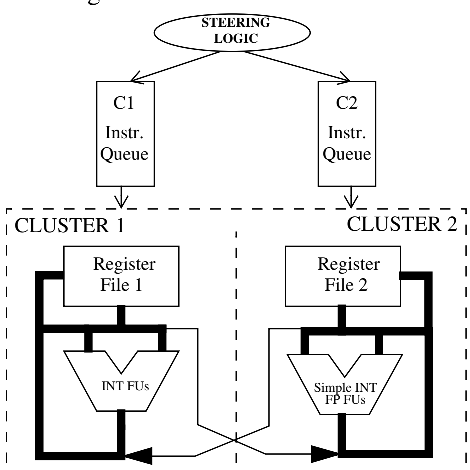

Figure 1: Processor architecture

2.1.Microarchitecture

The processor consists of two clusters, each of which contains a register file and some basic integer and logic functional units (see Figure 1). Moreover, one cluster contains complex integer functional units (multiplier and divider) and the other contains the floating-point functional units. Local bypasses within a cluster are responsible for forwarding result values produced in the cluster to the inputs of the functional units in the same cluster. Local bypassing is accomplished in effective 0 cycles, so one output in cycle i can be the input of a FU the following cycle ($i+1$). Inter-cluster bypasses are responsible for forwarding values between functional units residing in different clusters. Because inter-cluster bypasses require long wires, it is likely that these bypasses will be relatively slower and take one cycle or more in future technologies. With an adequate steering logic, local bypasses are used much more frequently than inter-cluster bypasses, and therefore the penalty of long bypasses is reduced.

The processor implements dynamic register renaming by means of a physical register file in each cluster and a single register map table. In this architecture, most integer instructions may be steered to either cluster, therefore their destination may be mapped to either physical register file. When a source operand of an instruction resides in the remote cluster, the dispatch logic inserts a "copy" instruction. This instruction will read the register when it is available and it will send the value through one of the inter-cluster bypasses. The forwarded value will be copied to a physical register in the requesting cluster for future reuse. Therefore, each time a copy instruction is dispatched, the forwarded logical register becomes mapped in two physical registers, one in each cluster, until it is remapped by a subsequent instruction. This scheme maintains some degree of register replication (on average, 3 registers per cycle) which dynamically adapts to the program requirements and is lower than replicating the whole integer register file [6]. Compared with a full replication scheme, it has also less communication requirements, and thus, less bypass paths and less register file write ports.

The renaming table is managed as follows: when an instruction is decoded, a new physical register is allocated for its output (if any) in the cluster it is sent to, and the register map table entry is updated accordingly. At the same time, the instruction keeps the old value of the renaming entry till it is committed. If the instruction is flushed from the pipeline (it was mis-speculated) the old value of the renaming entry is restored, and the allocated register is released. On the other side, if the instruction is committed, the old renaming physical register (or registers) is released. One logical register will only have two physical registers if its value has had to be copied from one cluster to the other. This scheme ensures that both clusters do not have different values of the same logic register.

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 and then its value is forwarded to the disambiguation hardware. Since both clusters have adders both clusters can execute effective-address computations. In the common disambiguation hardware, a load 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. Stores are issued at commit time.

2.2. Steering logic

There are two options on how to distribute the instructions between the two clusters: at compile -time (statically) or at run-time (dynamically). A static partitioning means that the compiler tags each instruction according to the cluster in which it will be executed. The main advantage is that it requires minimal hardware support. The drawbacks of this scheme are that the ISA of the processor has to be extended (for example, with one bit that indicates the cluster in which the instruction will be executed) and this implies that all the applications that run on this processor have to be recompiled. The second drawback is that deciding where an instruction will be executed long before it is in the pipeline is not as effective as taking this decision inside the processor. In the dynamic scheme on the other side, the ISA has not to be changed and therefore the clustering of a processor is transparent to the applications running on it. The dynamic scheme is also more adaptable to the state of the pipeline since it can decide where an instruction will be executed according to the actual state of the pipeline. Therefore the dynamic approach can minimise the number of communications and can balance the workload better than the static approach since the information used to perform the steering is obtained straight from the pipeline and not from an estimation of the compiler.

Clustered architectures introduce a trade-off between communication and workload balancing. On one side, we would like to have all dependent instructions in the same cluster in order to minimise the communications between clusters. On the other, we would like to have the same number of instructions in each cluster in order to maximise the use of the functional units. A good steering algorithm has to find the optimal trade-off between communication and workload balance.

All the communications are due to the fact that one of the operands (or both) is in the other cluster than the one where the instruction is executed. In order to minimise the communications between clusters, the steering logic sends dependent instructions to the same cluster. Furthermore, some communications can be hidden if the instruction waiting for it is not just waiting for this operand to arrive from the other cluster (e.g. the other operand is still being calculated or being loaded from memory).

Workload balancing should be performed with minimal impact on the communication overhead. In the proposed schemes, workload balance is achieved in many forms. The first and naive approach consists of a random variable that assigns the instructions to one of the clusters. Thus, the instructions are steered to any of the clusters with the same probability. The second approach assigns the instructions to the least loaded cluster. This approach will tend to balance the workload between both clusters and therefore, it can potentially achieve a higher performance.

If the communication and the workload balance considerations bring to a different steering for the same instruction, the instruction is steered according to the criteria the scheme things is the most adequate.

There are many alternatives on how the workload balance is measured at run-time. Actually, there are two parameters that define a balance figure: the workload figure and the boundary between balance and imbalance.

An estimation of the workload of each cluster could be the number of instructions in its instruction queue. However, this measure does not take into account the parallelism among these instructions. For instance, if one cluster has many instructions but each depends on the previous one, it will be quite idle and it could accept more workload. Thus, a better estimation of the workload is the number of ready instructions in each cluster, which depends on both the number of instructions and the parallelism among them.

We will consider that the workload is balanced when both clusters have the same number of ready instructions; and thus, the difference in the number of ready instructions will be used as a measure of the imbalance. Note however, that when both clusters have more ready instructions than their issue width, both can operate at the maximum rate and thus, we will consider this scenario as balanced even though the number of ready instructions is different. The same happens when both clusters have less instruction than the issue width.

The balance is computed every cycle and the average for the last N cycles is used by the steering logic. Actually, when implementing it, we just need the accumulated value since comparing the average to a certain threshold is the same as comparing the accumulated value to a threshold N times bigger. Adequate values of N (number of cycles used to average the balance figure) and the balance threshold are empirically determined in Section 3.

2.2.1. Steering schemes

The first steering scheme we have considered is quite simple (Simple Register Mapping Based Steering -RMBS). This scheme tries to keep the communications to a minimum since it sends the instructions to the cluster in which they have their register source operands. No consideration of balance is taken relying on the fact that the random steering that is used when the communication overhead is the same for both clusters is good enough to keep the balance steady. The algorithm works as follows:

- If the instruction has no source operands it is randomly assigned to a cluster.
- If the instruction has one source operand it is assigned to the cluster in which the operand is mapped.
- If the instruction has two source operands and both are in the same cluster it is sent to the same cluster,

otherwise it is randomly sent to one of the clusters.

This scheme tries to minimise the number of communications since it always sends the instructions where its operands are mapped. Communications are needed just in the case when one instruction has one operand in each cluster.

The second scheme (Balanced RMBS) includes some balance considerations. Whenever there is no preferred cluster from the point of view of communication overhead, the balance is taken into account and the instruction is sent to the least loaded cluster.

- If the instruction has no source operands it is assigned to the least loaded cluster.
- If the instruction has one source operand it is assigned to the cluster in which the operand is mapped.
- If the instruction has two source operands and both are in the same cluster it is sent to the same cluster; otherwise it is assigned to the least loaded cluster.

This scheme will improve significantly the workload balance while trying to keep the communications to a minimum since the balance is just taken into account whenever both clusters are considered equally good from the communications point of view.

The last steering scheme (Advanced RMBS) is the Balanced RMBS with a higher emphasis in the workload balance. This approach may decide that an instruction executes in a different cluster from the one in which it has its operands due to the poor balance at that moment. On one hand, this scheme might increase the number of communications between the clusters but on the other hand, it improves the workload balance. The algorithm is the following:

- If the there is significant imbalance the instruction is assigned to the least balanced cluster.
- Otherwise, it does the same as the Balanced scheme.

This scheme checks whether the imbalance value has exceeded a given threshold and in this case it sends the instruction to the cluster that most favours the balance. This scheme tries to achieve a higher workload balance at the cost of some extra communications.

For comparison purposes we introduce the modulo steering. This scheme consists of sending alternatively one instruction to each cluster if the instruction can be executed in both clusters. If it can just be executed in one of the clusters (integer multiplication/division or a FP computation) it is steered to that cluster. This scheme is simple and very effective regarding workload balance, there is also no consideration on communication so it might enforce many communication overhead.

3. Performance evaluation

3.1. Experimental framework

We have used a cycle-based timing simulator based on the SimpleScalar tool set [1] for performance evaluation. We extended the simulator to include register renaming through a physical register file and the steering logic described in Section 2. We used programs from the SpecInt95 [19] and Mediabench [9] to conduct our evaluation (see Table 1 and Table 2). All the benchmarks were compiled with the Compaq-DEC C compiler for an AlphaStation with the -O5 optimization level. For each benchmark, 100 million instructions were run after skipping the first 100 million; the Mediabench programs were run till completition. Six configurations have been simulated. First, the base architecture in which there are two conventional 4-way issue clusters where one executes integer operations and the other just floating-point operations. The second is the clustered architecture presented in this work with a modulo steering. The next three configurations are also based on this clustered architecture with the three proposed steering schemes: Simple RMBS, Balanced RMBS and Advanced RMBS. The main architectural parameters for these architectures are shown in Table 3

Finally, as an upper bound of the performance that can be achieved by the clustered architecture we consider an architecture (UB architecture) that is the base architecture with twice the issue width and twice the number of functional units. That is 8-way issue for integer plus 8-way issue for fp. This architecture will execute all integer instructions in the 8-way issue integer datapath without incurring in any communication penalty.

3.2. Results

In this section, we present results for the effectiveness of the steering algorithms and the performance improvement over the base architecture. In addition, we compare the results of the proposed steering schemes to the ones of the upper-bound architecture. The results are presented as follows: first a general overview of the performance of the different architectures except from the advanced RMBS is shown.

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 1: Benchmark programs SpecInt95

Benchmark	adpcm		epic	
	encode	decode	compresion	decompresion
Input	clinton.pcm	clinton.adpcm	test_image.pgm	test_image.E

Table 2: Benchmark programs MediaBench

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 intALU + 1 mul/div	3 intALU + 3 fpALU + 1 fp mul/div
	1 comm/cycle to C2	1 comm/cycle to C1
	Communications consume issue width	
Issue mechanism	4 instructions	4 instructions
	Out-of-order issue 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: Machine Parameters (separating cluster 1 and cluster 2 if not common)

Then, the reasons for the different performance levels are analysed. Afterwards, a motivation for the advanced algorithm is presented and its performance is determined. At the end, the performance of floating-point programs is evaluated and the advanced RMBS is compared to a static approach.

Figure 2 shows the IPC for each of the benchmarks and the harmonic mean for the SpecInt95, Mediabench and both together. The modulo scheme has no performance improvement (2% average slow-down) although it has a maximum speed-up of 17% in one of the benchmarks. In contrast, the Simple RMBS has significant wins (22% average speed-up) due to its much lower communication overhead, as we will see in short. The Balanced RMBS steering scheme reaches a 27% average speed-up due to its balance considerations. However, the speed-up of this scheme is significally under that of the UB architecture (44% speed-up over the base architecture) since the UB architecture does not have communication penalties. Below, we analyse the factors that influence the performance of the three steering schemes presented in Figure 2.

3.2.1. Inter-cluster communication

Figure 3 shows the average number of communications per executed instruction. As expected, we can see that the modulo scheme has a much larger number of communications than the other schemes (almost 23% of the instructions executed require inter-cluster communications), because it does not have any communication locality criterion. This huge communication overhead explains why it performs worse than the other schemes (see Figure 2). The other two schemes produce a similar number of communications since they use very similar communication criteria. We conclude that it is important to reduce the number of communications, since they cause overheads which significantly reduce performance, even for a 1-cycle communication latency.

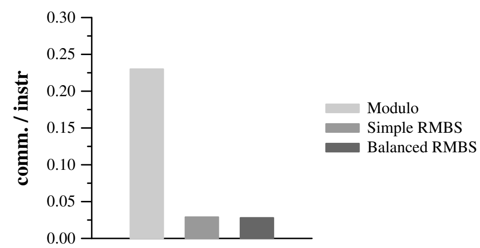

Figure 3: Average number of communications per dynamic instruction

The communication overhead depends also on the number of buses between the clusters. This paper shows the results for one bus each way (one from C1 to C2, and one from C2 to C1). We have also simulated the same architecture with 3 busses each way. The results show that the modulo scheme achieves an average speed-up of 5% over the base architecture, and that the other schemes do not change their performance (due to the low number of communications).

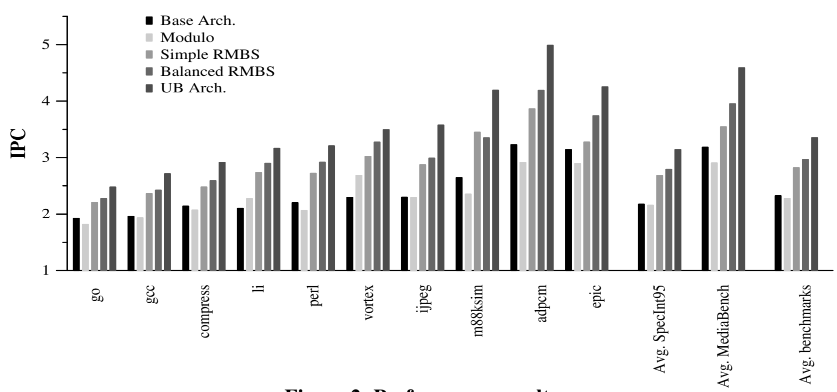

Figure 2: Performance results

3.2.2. Workload balance

Figure 4 presents the distribution function of the workload balance figure for a particular program (ijpeg). Remember that the figure is an average and that it only considers imbalance as the situation where one cluster has more ready instructions than the issue width and the other has fewer. The peak at zero is the percentage of cycles that there was not imbalance and the rest is the distribution of the figure for the remaining cycles. This distribution is similar in all the benchmarks examined. We can see that the least balanced is the Simple RMBS since it does not implement any balance policy. The Balanced RMBS has a better workload balance due to the balance considerations it implements. The most balanced scheme is the modulo, since it sends one instruction to each cluster alternatively (see Section 2), and this policy produces a near optimal balance.

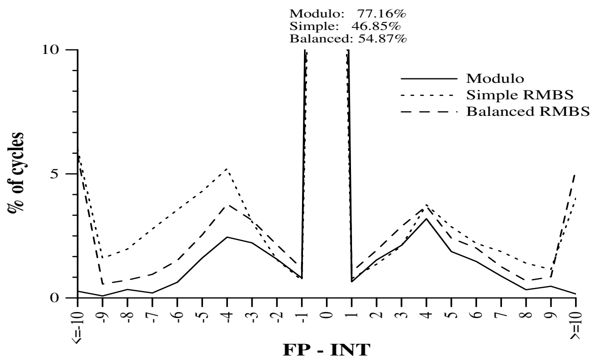

Figure 4: Workload balance figure distribution (ijpeg)

Figure 5 shows the average IPC of the three steering schemes plus the base and UB architectures for a 0-cycle communication latency. Since there is no communication overhead in this scenario, this Figure illustrates the impact on performance of the workload balance. In accordance with the results in Figure 4, the Simple RMBS has the lowest performance (13% lower than the UB architecture), the Balanced RMBS performs better (just 7% lower than the UB architecture), and the modulo scheme achieves the best performance (just 4% lower than the UB architecture).

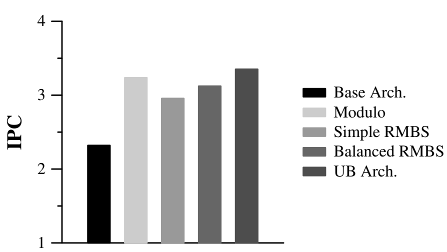

Figure 5: IPC for a 0-cycle communication latency

Overall, we can conclude that the modulo scheme performs worse than the others (see Figure 2) because, although it has a better workload balance, it enforces many more communications. At the same time, the Balance scheme performs better than the Simple one (see Figure 2) because it has a better workload balance while having a similar number of communications. We can conclude that this scheme achieves the best trade-off between communication and workload balance.

3.2.3. Improving the workload balance

In Figure 2, we can see that the speed-up of the Balance RMBS is 27% while that of the UB architecture is 44%, which suggests that there is still some potential improvement. This figure motivated the proposal of a new scheme (Advanced RMBS, see Section 2) that tries to improve the workload balance. This scheme tries to avoid strong imbalances. In consequence, when the balance exceeds a certain threshold, the scheme will give priority to optimising the balance rather than the communications.

Some experiments have been conducted in order to find out the best parameters that define the workbalance measure. As explained in Section 2, the workload balance is measured as the average imbalance for the last N cycles (history size). We have tried several configurations with different thresholds and history sizes. Experimentally, it has been found that mid-range (6 to 10) thresholds perform better since low boundaries (below 6) tend to increase the number of communications and high thresholds (above 10) tend to diminish the effect of the technique. Consequently, the threshold for the rest of the experiments is assumed to be eight instructions. This experiments also shows that the performance mainly varies with the history size. Figure 6 shows the performance of the Advanced scheme compared to the Balanced scheme for several history sizes. From Figure 6, we can observe that having extra balance

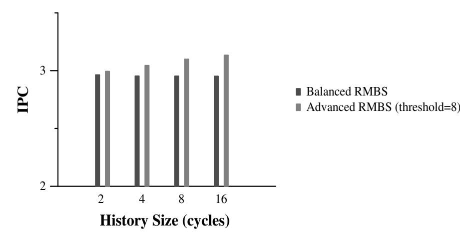

Figure 6: Performance of the Advanced RMBS(average)

considerations, as the Advanced scheme does, improves performance. We can also see that the longer the history the better the performance, for the range considered. This is due to the fact that long histories minimise the effect of eventual imbalances.

Figure 7 shows the performance of the Advanced RBMS compared to the other schemes. This scheme shows the best performance (35% average speed-up) of the steering schemes presented. Its speed-up is very close to that of the upper bound architecture (44% average), even though the Advanced RMBS has one cycle penalty for each communication.

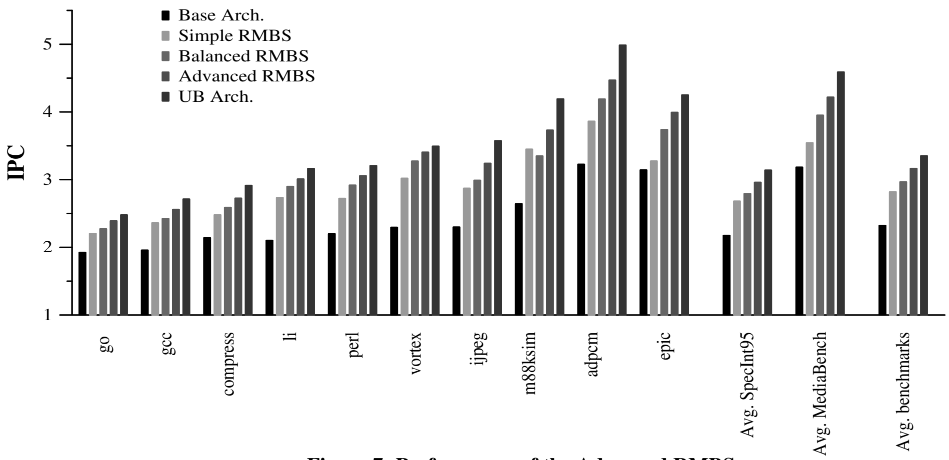

Figure 7: Performance of the Advanced RMBS

3.2.4. Impact of the communication latency

In future architectures, communications latencies will likely be longer than 1 cycle [11]. Therefore, we have analysed the impact of the communication latency on the performance. Figure 8 shows the performance of the schemes presented in this work for a range of communication latencies between 0 and 10 cycles. The results for a 0 cycle latency and 1 cycle latency are the same as those in Figure 5 and Figure 7 respectively.

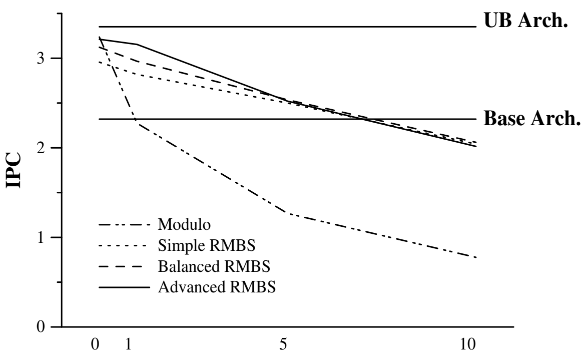

Figure 8: Performance for different communication latencies (average)

First, we can observe that as latency increases there is a growing performance loss and this loss is much greater for the modulo scheme than for the others since it does not take into account the communications. This behaviour stresses the increasing importance of minimising the number of communications. On the other side, the impact of a higher latency in the other three schemes is very similar since they have similar criteria to avoid communication. However, we can see significant differences among them for small latencies due to the different balance considerations they implement (as shown in the previous section).

For high latencies, the performance of all the schemes presented in this work is quite below the UB architecture. In this scenario, the steering schemes should make more emphasis in reducing the number of communications, even at the expense of a worse workload balance. In the future, we plan to investigate steering schemes oriented to this scenario.

3.2.5. Applicability to floating-point programs

We have evaluated whether sending integer instructions to the FP cluster might degrade the performance of floating-point programs due to the sharing of resources in that cluster. We have measured the performance of the Advanced RMBS and the base architecture for several SpecFP95 [19] benchmarks. Figure 9 shows the speed-ups relative to the base architecture.

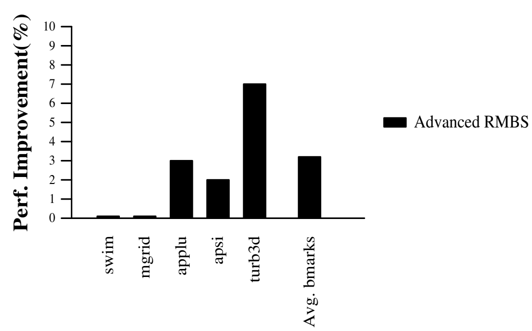

Figure 9: Performance of the SpecFP95

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.

3.2.6. Dynamic versus static steering

In Figure 10, the performance of the Advanced RMBS is compared with the results presented by Sastry, Palacharla and Smith [16]. In that work, the authors presented a compile-time partitioning algorithm for a similar architecture to the one of this paper. The results have been obtained with the same experimental framework described in that paper (SimpleScalar based simulator with the same architectural parameters, benchmarks, compiler and inputs). Figure 10 depicts the speed-ups of both architectures..

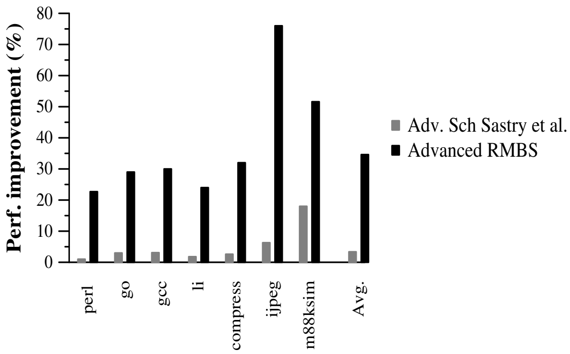

Figure 10: Speed-up over the base architecture

In this case, the dynamic approach is much more effective than the static one (the improvement achieved over the base architecture is 10 times bigger) for several reasons. First, because the dynamic approach not only tries to reduce inter-cluster communication but also workload imbalance, which was already reported by Sastry, Palacharla and Smith to be one of the main drawbacks of their approach. Second, since the Advanced RMBS steers instructions dynamically, it adapts more accurately to many run-time conditions that are difficult to estimate at compile time.

4. Related work

The steering schemes presented in this paper are targeted for a specific superscalar microarchitecture inspired in the proposal of Sastry, Palacharla and Smith [16]. Our work differs in that they proposed a static partitioning scheme, based on an extension of the concept of the “load/store slice” (which consists of all the instructions that compute addresses, and their predecessors in the data dependence graph [14]). Another difference is that our architecture does not constrain address computations to be dispatched to a specific cluster, so it allows a more flexible partitioning. While borrowing the main advantages of their architecture, our steering scheme largely outperforms their partitioning approach.

Other closely related approaches are the Dependence-based [13], the Pews [8] and the Multicluster [2] architectures. In these proposals, the microarchitecture partitions datapaths, functional units, issue logic and register files into symmetrical clusters. Our work, instead, is based on a slight modification of a conventional architecture that converts the FP unit in a second cluster available for integer computations. Their steering schemes are also different as outlined below.

In the Dependence-based paradigm, instructions are steered to several instruction FIFO queues instead of a conventional issue window, according to a heuristic that ensures that two dependent instructions are only queued in the same FIFO if there is no other instruction in between. This heuristic lacks of any explicit mechanism to balance the workload, which is instead adjusted implicitly by the availability of free FIFO queues.

In the Pews architecture, the steering scheme is quite simple, since it always places an instruction in the cluster where the source operand is to be produced, except if the operands are to be produced in different clusters, in which case the algorithm tries to minimize the communication overhead (which is a function of the forwarding distance of the operands, in this ring-interconnected architecture). This algorithm also lacks of a workload balance mechanism.

In the Multicluster architecture, the register name space is also partitioned into two subsets, and program partitioning is done at compile time without any ISA modification, by the appropiate logical register assignment of the result of each instruction. Both the workload balance and inter-cluster communication are estimated at compile time. The same authors proposed a dynamic scheme [3] that adjusts run-time excess workload by re-mapping logical registers. However, they found most heuristics to be little effective since the re-mapping introduces communication overheads that offset almost any balance benefit.

Another related architecture is the decoupled one [17]. In this case, the partitioning is based on sending the effective address calculation of the memory accesses to a cluster and the remaining instructions to the other. This partitioning is similar to the load/store slice proposed by Palacharla et al. [14].

Clustering is also present in some multithreading architectures. These architectures execute threads that are generated either at compile-time (Multiscalar processors [5, 18] among others) or at run-time (Trace Processors [15, 20]), Clustered Speculative Multithreaded Processors [10]). The criteria used to partition programs is based on control-dependences instead of data dependences. Besides, they make extensive use of data speculation techniques. Clustering can also be applied to VLIW architectures [4, 12], but they perform cluster assignment at compile-time.

There are several techniques to improve the performance of multimedia programs. The architecture presented in this article is not targeted especially to this kind of programs but to general purpose ones. Nevertheless, the performance improvement achieved with this architecture is significant in this kind of applications and has the advantage of improving the performance in any program, not just in the multimedia ones (as multimedia extensions do). There is also a parallelism between some early MMX extensions [7] and the presented architecture since the fp cluster is extended with integer instructions in both clusters. Nevertheless, MMX extensions include a SIMD execution model that is applied to vectorizable code.

5. Conclusions

In current superscalar processors, all floating-point resources are idle during the execution of integer programs. This problem can be alleviated if the floating-point cluster is extended to execute integer instructions and these are dynamically sent to one cluster or the other. The required modifications are minor and the resultant microarchitecture stays very similar to a conventional one. Furthermore, no change in the ISA is required. However, in this architecture there is a trade-off between workload balance and inter-cluster communication overhead and the steering logic is responsible for optimising it. We presented three steering schemes and evaluated them. The performance figures showed an average speed-up of 35% over a conventional 8-way issue (4 int + 4 fp) machine. Hence, with minimal hardware support and no ISA change, idle floating-point resources on current superscalar processors can be profitably exploited to speed-up integer programs.

Acknowledgments

This work has been supported by the Ministry of Education of Spain under contract CYCIT TIC98-0511-C02-01 and by the European Union through the ESPRIT program under the MHAOTEU (EP 24942) project. The research described in this paper has been developed using the resources of the CEPBA. Ramon Canal would like to thank his fellow PBCs for their patience and precious help.

References

[1] D. Burger, T.M. Austin, S. Bennett. “Evaluating Future Microprocessors: The SimpleScalar Tool Set”, *Technical Report CS-TR-96-1308*, University of Wisconsin-Madison, 1996.

[2] K.I.Farkas, P.Chow, N.P.Jouppi, Z.Vranesic. “The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, in *Proc of the 30th. Ann. Symp. on Microarchitecture*, December 1997, pp. 149-159.

[3] K.I.Farkas. “Memory-system Design Considerations for Dynamically-scheduled Microprocessors”, *PhD thesis*, Department of Electrical and Computer Engineering, Univ. of Toronto, Canada, January 1997.

[4] M.M. Fernandes, J.Llosa and N.Topham, “Distributed Modulo Scheduling”, in *Proc of the 5th. Int. Symp. on High Performance Comp. Arch.*, January 1999, pp. 130-134.

[5] M. Franklin, “The Multiscalar Architecture”, *Ph.D. Thesis*, *Technical Report TR 1196*, Computer Sciences Department, Univ. of Wisconsin-Madison, 1993.

[6] L. Gwennap. “Digital 21264 Sets New Standard”, *Microprocessor Report*, 10 (14), Oct. 1996.

[7] L. Gwennap. “Intel’s MMX Speeds Multimedia Instructions”, *Microprocessor Report,* 10(3), March 1996.

[8] G.A.Kemp, M.Franklin, “PEWs: A Decentralized Dynamic Scheduler for ILP Processing”, in *Proc. of the Int. Conf. on Parallel Processing*, August 1996, pp. 239-246.

[9] C. Lee, M. Potkonjak and W. H. Mangione-Smith, “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, *Proc. of the IEEE/ACM Int. Symposium on Microarchitecture* (Micro 30), December 1997, pp. 330-335.

[10] P. Marcuello and A. González, “Clustered Speculative Multithreaded Processors”, *Proc of the 13th ACM Int. Conf. on Supercomputing*, June 1999, pp. 365-372.

[11] D.Matzke, "Will Physical Scalability Sabotage Performance Gains", *IEEE Computer Vol. 30, num. 9*, September 1997, pp. 37-39.

[12] E.Nystrom and A.E.Eichenberger, “Effective Cluster Assignment for Modulo Scheduling”, in *Proc of the 31st. Ann. Symp. on Microarchitecture*, pp. 103-114.

[13] S. Palacharla, N.P. Jouppi, and J.E. Smith, “Complexity-Effective Superscalar Processors” in *Proc of the 24th. Int. Symp. on Comp. Architecture*, June 1997, pp. 1-13.

[14] S.Palacharla, J.E.Smith “Decoupling Integer Execution in Superscalar Processors”, in *Proc. of the 28th. Ann. Symp. on Microarchitecture*, November 1995, pp. 285-290.

[15] E.Rotenberg, Q.Jacobson, Y.Sazeides and J.E.Smith, “Trace Processors”, in *Proc of the 30th. Ann. Symp. on Microarchitecture*, December 1997, pp. 138-148.

[16] S.S.Sastry, S.Palacharla, J.E.Smith, “Exploiting Idle Floating-Point Resources For Integer Execution”, in *Proc. of the Int. Conf. on Programming Lang. Design and Implementation*, June 1998, pp. 118-129.

[17] J.E. Smith, “Decoupled Acces/Execute Computer Architectures”, *ACM Transactions on Computer Systems,* 2(4), November 1984, pp. 289-308.

[18] G.S.Sohi, S.E.Breach, and T.N.Vijaykumar, “Multiscalar Processors”, in *Proc. of the 22nd Int. Symp. on Computer Architecture*, June 1995, pp. 414-425.

[19] Standard Performance Evaluation Corporation, *SPEC Newsletter*, September 1995.

[20] S. Vajapeyam and T. Mitra, “Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences”, in *Proc. of the Int. Symp. on Computer Architecture,*June 1997, pp. 1-12.