Dynamic Cluster Assignment Mechanisms

Ramon Canal, Joan Manuel Parcerisa and Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain
{rcanal,jmanel,antonio}@ac.upc.es

Abstract

Clustered microarchitectures are an effective approach to reducing the penalties caused by wire delays inside a chip. Current superscalar processors have in fact a two-cluster microarchitecture with a naive code partitioning approach: integer instructions are allocated to one cluster and floating-point instructions to the other. This partitioning scheme is simple and results in no communications between the two clusters (just through memory) but it is in general far from optimal because the workload is not evenly distributed most of the time. In fact, when the processor is running integer programs, the workload is extremely unbalanced since the FP cluster is not used at all. In this work we investigate run-time mechanisms that dynamically distribute the instructions of a program among these two clusters. By optimizing the trade-off between inter-cluster communication penalty and workload balance, the proposed schemes can achieve an average speed-up of 36% for the SpecInt95 benchmark suite.

Keywords: Clustered microarchitectures, dynamic code partitioning, steering logic, dynamically scheduled processors.

1. Introduction

Scaling-up current superscalar microarchitectures will face significant problems such as the growing impact of wire delays [2] [13] and increasing complexity of some parts such as the issue and rename logic [15]. Clustering is an effective solution to these problems. A clustered microarchitecture partitions some of this critical logic into simpler parts and, at the same time, it reduces the impact of wire delays by keeping most of the communications local to single clusters and avoiding communications among different clusters whenever possible.

Current superscalar processors are in fact partitioned into two clusters, one for integer instructions and the other for FP operations. Each of these clusters has its own instruction queue[1], issue logic, functional units and register file. However, these two data-paths can be underutilized due to a poor workload balance. This is especially true when the processor is running integer applications, which (almost) only use the integer data-path. Providing the two data-paths with the capability of executing any type of instructions would imply that every cluster should have any type of functional unit. However, since FP applications are rich in integer instructions, Palacharla and Smith [16] addressed this drawback by proposing a more cost-effective approach based on extending the FP data-path with functional units for simple integer and logical operations, which are usually the most frequent instructions. This requires minor modifications to existing microarchitectures and may result in significant speed-ups, especially for integer applications, since both data-paths can process instructions in parallel.

Deciding which instructions are executed in each cluster is a critical issue of clustered microarchitectures. We will refer to this problem as code partitioning. This work focuses on code partitioning mechanisms for clustered microarchitectures. Although the schemes presented are evaluated in one architecture, we believe that the same schemes can be used in more generic clustered architectures. We first show that dynamic mechanisms can be more effective than static ones. Then, we propose several dynamic mechanisms and evaluate their performance. We report average speed-ups of up to 36% for the SpecInt95 benchmark suite, when compared with a conventional microarchitecture with the naive integer-FP

1. In some processors the instruction queue may be shared.

code partitioning. We also show that the proposed approaches outperform previously proposed dynamic schemes.

The rest of this paper is organized as follows. Section 2 describes the assumed processor microarchitecture. Section 3 presents and evaluates several code partitioning mechanisms. It also includes comparisons with previously proposed mechanisms. Section 4 discusses some related work. Finally, Section 5 summarizes the main conclusions of this work.

2. Processor Microarchitecture

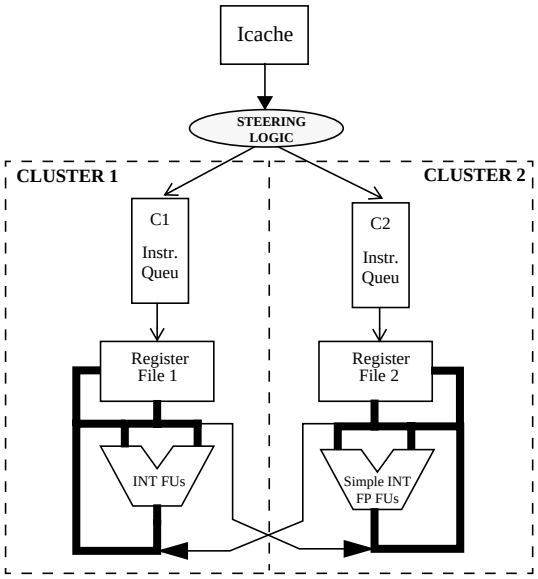

Figure 1: Processor architecture

The target processor microarchitecture is based on the proposal made by Palacharla and Smith [16] and also investigated by Sastry, Palacharla and Smith [18], which extends a conventional microarchitecture in order to allow simple integer and logic instructions to be executed in both clusters. Unlike those works, we propose a dynamic code partitioning scheme, which is performed by a hardware that is referred to as steering logic. Dynamic partitioning has the advantage of not requiring any modification to the ISA, unlike the static approaches. Furthermore, the complexity of this hardware is low, as will be later shown. Basically, it requires some modifications in the register renaming mechanism and some tables that hold information used by the partitioning heuristics.

Figure 1 shows a block diagram of the processor architecture. The main differences with a conventional architecture are the steering logic and the buses that allow values to be copied from one cluster to the other. Instructions are fetched and decoded by a centralized hardware and then, they are dispatched to one of the two clusters. Each cluster has its own data-path and instruction issue logic. Both clusters have a set of simple integer and logic functional units. In addition, one cluster contains complex integer functional units (multiplier and divider) and the other has floating-point functional units. Local bypasses are responsible for forwarding result values produced in a cluster to the inputs of the functional units in the same cluster. A local bypass takes *0* cycles, so an output in cycle *i* can be an input of a FU the following cycle (*i+1*). Inter-cluster bypasses are responsible for forwarding values between different clusters. Therefore, they are slower than local ones, and we assume that they take one cycle.

Dynamic register renaming is performed by means of a physical register file in each cluster and a single register map table. Since integer instructions can be executed in both clusters, the entries of the map table for integer registers contain two fields that identify the mapping in each cluster. When an instruction is decoded, the steering logic decides in which cluster it is to be executed and a physical register from that cluster is allocated for the destination operand (if any). When a source operand of an instruction resides in the remote cluster, a physical register in the local cluster is allocated and the dispatch logic inserts a “copy” instruction that will move the data from the remote to the local cluster. This instruction will read the operand when it is available and will send the value through one of the inter-cluster bypasses. Copy instructions compete for issue slots and processor resources (e.g. register file ports) as any other instruction. This scheme implies some register replication but only for values that are used in the two clusters. We will show that the proposed partitioning schemes incur in a low degree of replication.

Load and store instructions are internally split into two operations, one for computing the effective address and another that performs the memory access. Address calculation can be performed in any of the two clusters since it involves a simple integer operation (addition). Then, the instruction is forwarded to a unique disambiguation logic that decides when the instruction can perform its memory access. A load reads from memory after being disambiguated with all previous stores, whereas stores write to memory at commit.

3. Code Partitioning Schemes

This section presents several partitioning approaches and evaluates their performance. We first define some terminology and describe the experimental framework. Then, we compare the effectiveness of static partitioning

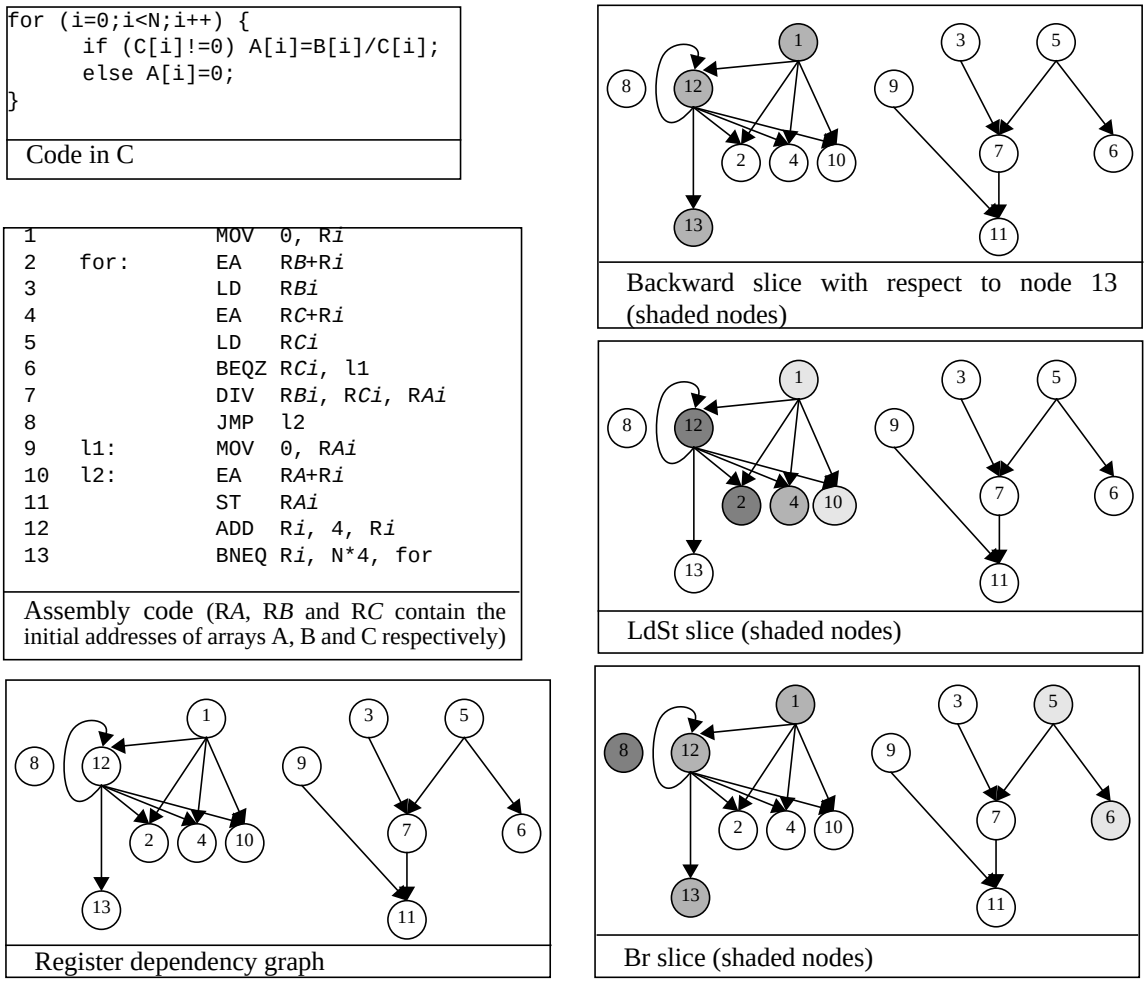

Figure 2: Example of a RDG

versus a simple dynamic mechanism. Next, alternative dynamic schemes are presented and evaluated. Finally, a comparison with a previously proposed dynamic scheme is presented.

3.1. Terminology

The 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 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.

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* [18]. Figure 2 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 2 shows the LdSt slice and the Br slice of the sample program. Each backward slice is shaded in a different gray level.

We will refer to the cluster of the processor that can perform just integer operations as the *integer cluster*, and the cluster that can execute FP and simple integer instructions will be called the *FP cluster*.

3.2. Experimental Framework

Performance figures were obtained through a cycle-based timing simulator based on the SimpleScalar tool set v3.0 [3], which was extended to simulate the architecture described in section 2. Results are presented for the SpecInt95 benchmark suite. Table 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 2 shows the architectural parameters of the assumed processor.

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 2 except that it has neither integer units in the FP cluster nor inter-cluster bypasses.

3.3. Static versus Dynamic Partitioning

A static partitioning approach requires some extensions to the ISA in order to allow the compiler to specify to the hardware the target cluster for each instruction. Moreover, it is less flexible than a dynamic approach since all dynamic instances of the same static instruction are executed in the same cluster. On the other hand, its hardware complexity is negligible.

The static partitioning proposed by Sastry *et al.* [18] is based on sending to the integer cluster all instructions that belong to the subgraph defined by the LdSt slice, probably extended with neighbor instructions. This extension is based on some heuristics that try to approximate its effect in terms of workload balance and communication overheads.

Figure 3 compares the speed-ups of that static partitioning with the speed-ups achieved by a simple dynamic partitioning 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 partitioning scheme as *LdSt slice steering*. The numbers for the static partitioning have been obtained from the original paper [18] 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 partitioning speed-up is just 3%.

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 int mul/ div	3 intALU + 3 fpALU + 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 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 2: Machine parameters (split into cluster 1 and cluster 2 if not common)

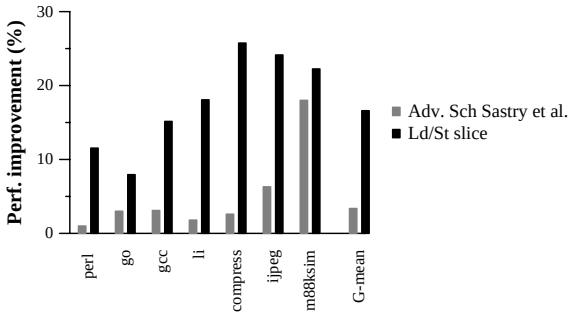

Figure 3: Static versus dynamic partitioning

.

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: Benchmarks and their inputs

This dynamic partitioning 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.

3.4. LdSt Slice Steering versus Br Slice Steering

The performance of any partitioning 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 partitioning 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 suggest an alternative partitioning 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 for the LdSt slice steering.

Figure 4 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 5. 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. The workload of a cluster can be measured as the number of ready instructions it has. Figure 6 shows the distribution function of the workload balance (that is, the difference between the number of ready instructions in each cluster

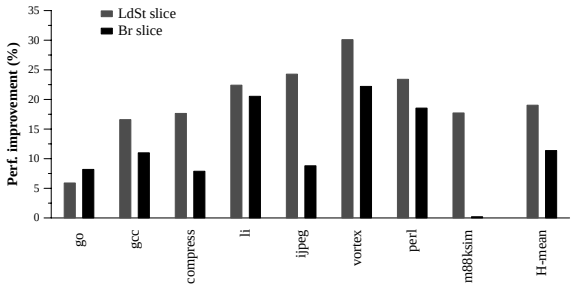

Figure 4: LdSt slice versus Br slice steering

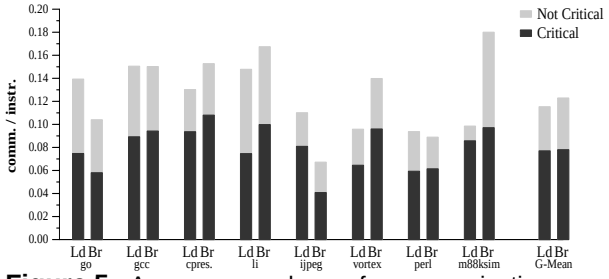

Figure 5: Average number of communications per dynamic instruction for slice steering

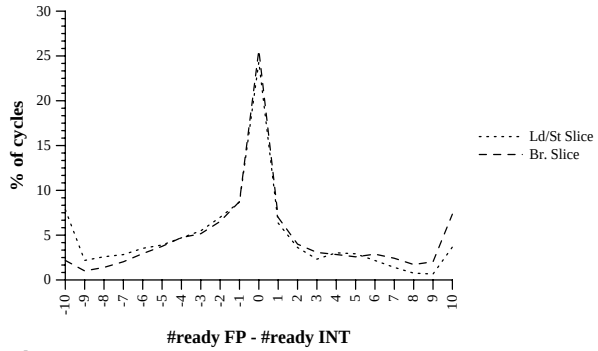

Figure 6: Distribution of the diference in the number of ready instructions between each cluster (SpecInt95 average)

for each cycle). It can be seen that both dynamic partitioning 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 (resp. Br slice) were dispatched to the integer cluster. This motivates the next partitioning scheme.

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 there is a strong workload imbalance (see next paragraph). 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.

The workload imbalance may be estimated by counting the difference in the number of instructions steered to each cluster (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 instruction 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 work considers the two metrics (I1 and I2) by maintaining a single integer imbalance counter that combines the two informations. Each cycle, the counter is updated according to the average of I2, computed along N cycles. It is also updated with I1, by incrementing or decrementing it for each instruction steered, so every instruction decoded in the same cycle sees a different value of the workload balance and thus, massive steerings to one cluster are avoided.

To determine whether there is a strong imbalance, the absolute value of this counter is compared with a given threshold. We have empirically determined that 16 and 8 are adequate values for N and the threshold respectively. 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, metric I1 alone gives performance figures quite close to those produced by the combination of I1 and I2.

Figure 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 8. This figure shows that the non-slice balance steering significantly increases the number of communications for

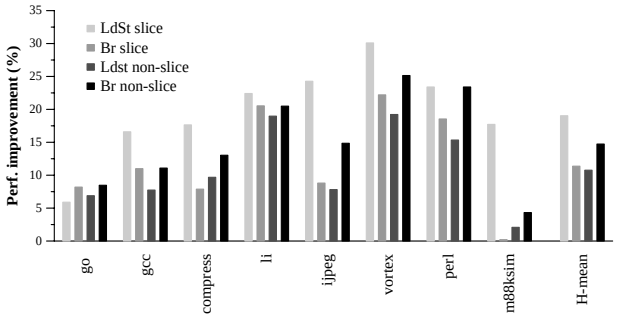

Figure 7: Non-slice balance steering versus slice steering

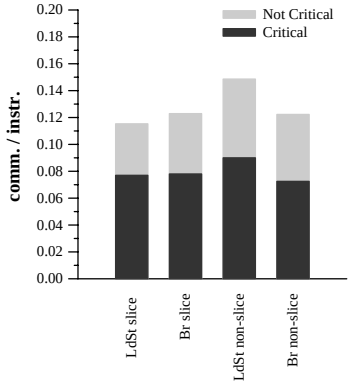

Figure 8: Average number of communications per dynamic instruction (SpecInt95 average)

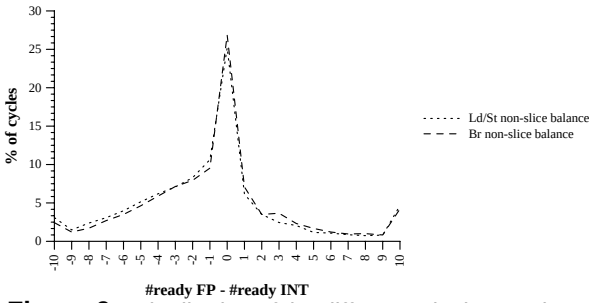

Figure 9: Distribution of the difference in the number of ready instructions between each cluster (SpecInt95 average)

the LdSt slice whereas it has about the same number of communications as the slice steering for the Br slice.

Figure 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 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 partitioning scheme.

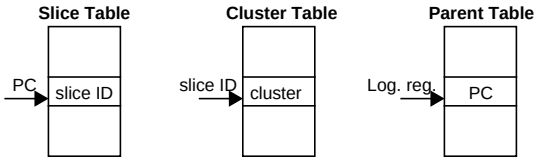

Figure 10: Hardware support for the slice balance steering

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 backwards 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 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 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 12 shows the distribution of the workload balance for the slice balance steering (LdSt and Br) and compares it with that of a naive

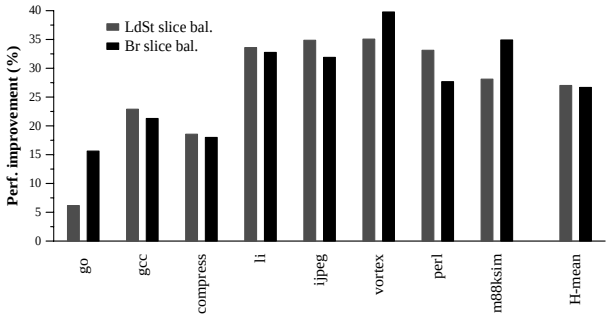

Figure 11: Slice balance steering performance

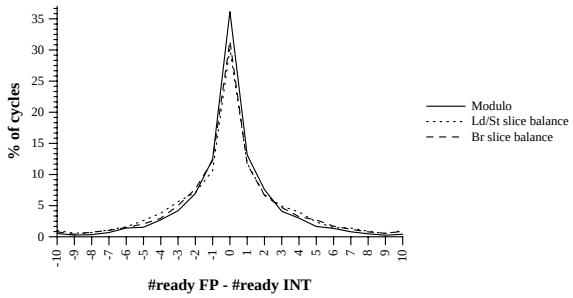

Figure 12: Distribution of the difference in the number of ready instructions between each cluster (SpecInt95 average)

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 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.

3.7. 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 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 slices. 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 3.5 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 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 13 shows the performance of the priority slice balance steering. It achieves an average speedup 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 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.

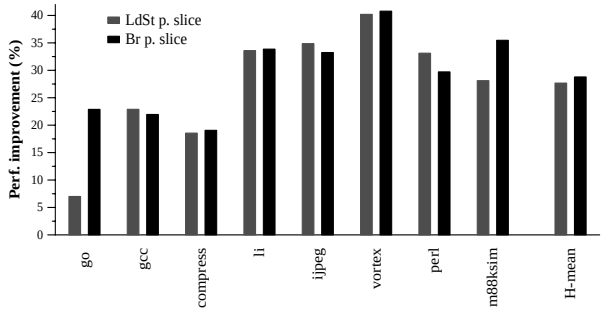

Figure 13: Priority slice balance steering performance

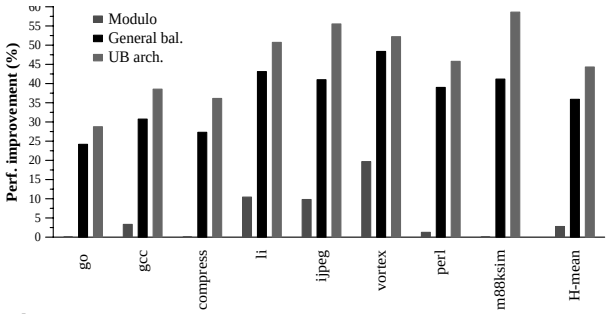

Figure 14: General balance steering

3.8. General Balance Steering

The last presented scheme is a particular case of the previous one, in which the criticality threshold is set so high that there are no critical instructions and 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 programs slices (see figure 10) is not needed and no extra hardware is required to detect the criticality of instructions.

Figure 14 shows the performance of this scheme. It also includes the performance of the modulo steering and that of a 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 partitioning 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 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).

As outlined in section 2, the cluster microarchitecture requires some degree of register replication. We have evaluated the average number of logical register that have

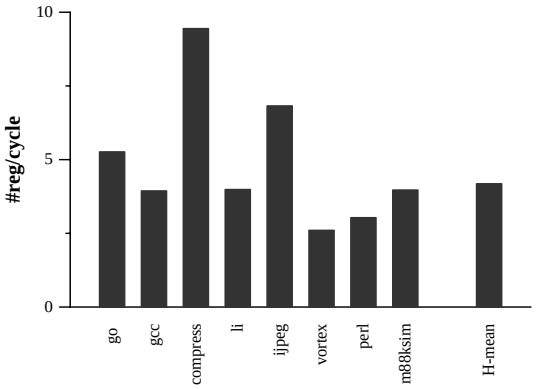

Figure 15: Register replication for the general balance steering

a physical register allocated in both clusters. Results show (see figure 15) that the required register replication is very low. Instead of replicating the whole physical register file as the Alpha 21264 processor does [10], on average this architecture requires 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 [7].

This scheme performs at the same level when there is just one bus each way to connect the clusters. Similar schemes to the General Balance one can be found in a work of the same authors [4].

3.9. Comparison with Other Dynamic Partitioning Approaches

Palacharla, Jouppi and Smith [15] recently proposed a dynamic partitioning approach for a different clustered architecture that could be also applied to our assumed architecture. The basic idea is to model each instruction queue as if it was a collection of FIFO queues with instructions capable of issuing from any slot within each individual FIFO. Instructions are steered to FIFOs following some heuristics that ensures that a FIFO only contains dependent instructions, each instruction being dependent on the previous instruction in the same FIFO (for more details refer to the original paper [15]). In this case, the FIFO approach has been implemented (8 FIFOs in each cluster and each 8-deep); and thus, it has been simulated with the same architecture and benchmarks used for the schemes presented in this work.

Figure 16 shows that the performance of the general balance steering described in the previous section 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.

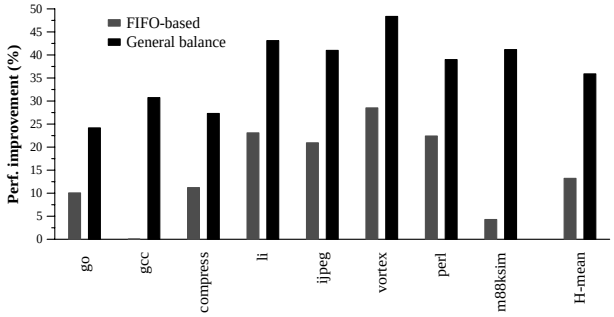

Figure 16: General balance steering versus FIFO-based steering [15]

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.

4. Related Work

The proposal of Sastry, Palacharla and Smith [18] and that of Palacharla, Jouppi and Smith [15] are two partitioning schemes that can be applied to the same type of architecture assumed in this work. The former is a static approach whereas the latter is based on run-time mechanisms. We have briefly outlined these techniques in sections 3.3 and 3.9 respectively and we have shown that the mechanisms proposed in this work, in particular the general balance steering, significantly outperform both of them.

Another clustered architecture with a mostly-static code partitioning with some run-time support to improve workload balance is the Multicluster architecture [6]. In this case, the processor consists of several identical clusters whereas our proposal focuses on an architecture that requires minor modifications to a conventional processor microarchitecture.

Kemp and Franklin proposed a clustered architecture [11] where instructions are assigned to clusters based on inter-instruction register dependencies. However, since they assume a centralized register file, the steering scheme only needs to group two dependent instructions in the same cluster when the value from the producer is not still available at the time the consumer is decoded. This simple steering scheme is not suitable for our "distributed" register file, and in addition, it does not address the load balancing problem.

Clustering can also be applied to VLIW architectures [8] [14]. In this case the partitioning is done at compile time.

Other authors have proposed clustered microarchitectures in which the partitioning scheme focuses on reducing the control dependence penalties. Examples of such architectures are the Multiscalar [9] [19], SPSM [5], Superthreaded [20], Trace Processors [17] [21], Speculative Multithreaded [12] and Dynamic Multithreaded [1]. In such architectures, each cluster executes a different thread of control, all except one being speculative. Partitioning to reduce branch penalties and data dependence penalties are orthogonal paradigms that attack different problems and the two would combine nicely.

5. Conclusions

We have proposed a number of run-time mechanisms that dynamically partition a sequential program into the different clusters of a clustered microarchitecture. We have focused on a two-cluster processor that is based on a conventional superscalar microarchitecture with the capability of executing simple integer operations in both the integer and the FP datapaths. Nevertheless, the schemes can be used in a generic clustered architecture with symmetric clusters.

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. We have shown that all the schemes provide a significant speed-up over a conventional microarchitecture. For instance, the general balance steering scheme achieves an average speed-up of 36% for the SpecInt95 and its IPC is just 8% below that of a conventional processor with twice its issue width. We have also shown that the proposed schemes significantly outperform previous dynamic and static proposals.

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 (EP24942) project. The research conducted in this paper has been developed using the resources of the CEPBA. Ramon Canal would like to thank his fellow PBC's, Sandra and Dídac for their patience and precious help.

References

[1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor", in *Proc. of the 31st. Int. Symp. on Microarchitecture*, pp. 226-236, 1998.

[2] M.T. Bohr, "Interconnect Scaling - The Real Limiter to High Performance VLSI", in *Proc. of the 1995 IEEE Int. Electron Devices Meeting*, pp. 241-244, 1995.

[3] 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.

[4] R. Canal, J.M. Parcerisa, A. Gonzalez, "A Cost-Effective Clusterd Architecture", in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques*, pp. 160-168, 1999.

[5] P.K. Dubey, K. O'Brien, K.M. O'Brien and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading", in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 109-121, 1995.

[6] 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, pp149-159

[7] K.I. Farkas, N.P. Jouppi and P. Chow. "Register File Considerations in Dynamically Scheduled Processors", in *Proc. of Int. Symp. on High-Performance Computer Architecture*, pp. 40-51, 1996

[8] M.M. Fernandes, J.Llosa and N.Topham, "Distributed Modulo Scheduling", in *Proc of the 5th. Int. Symp. on High Performance Comp. Arch.*, Orlando, Florida, Jan 1999. pp 130-134

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

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

[11] G.A.Kemp, M.Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing", in *Proc. of the Int. Conf. on Parallel Processing*. 1996, v.1, pp 239-246.

[12] P. Marcuello, A. González and J. Tubella, "Speculative Multithreaded Processors", in *Proc of the 12th ACM Int. Conf. on Supercomputing*, pp 77-84, July 1998.

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

[14] E.Nystrom and A.E.Eichenberger, "Effective Cluster Assignment for Modulo Scheduling", in *Proc of the 31st. Ann. Symp. on Microarchitecture*, Dallas, Texas, November 1998. pp. 103-114

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

[16] 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.

[17] E.Rotenberg, Q.Jacobson, Y.Sazeides and J.E.Smith, "Trace Processors", in *Proc of the 30th. Ann. Symp. on Microarchitecture*, 1997.

[18] 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.* Montreal, 1998.

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

[20] J-Y. Tsai and P-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation", in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35-46, 1996.

[21] 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*, pp. 1-12, 1997.