CHAPTER 2

EXPERIMENTAL ENVIRONMENT

In this chapter, the design of a typical clustered superscalar architecture, its default main parameters, and the experimental environment utilized along this thesis, including the simulation methodology and the benchmarks are described.

The clustered microarchitecture, which we will refer to as the Reference Clustered Architecture, will be used throughout the rest of this thesis both as a baseline for performance comparisons, and as an aid in describing our proposed techniques. It belongs to a well known class of architectures, and shares many similarities with other proposed architectures that have been studied and evaluated before ([29, 52, 53, 70, 73] among others, refer to section 1.2), but especially with the architecture proposed by V. Zyuban in his Ph.D. thesis [112], where he performs a thorough analysis of its power and performance efficiency. Therefore, it is not our purpose to repeat such evaluations, but to use it as a vehicle for studying and developing new techniques on top of it, with the goal of improving performance and reducing complexity.

2.1 Reference Clustered Microarchitecture

The assumed processor microarchitecture is a clustered implementation of an out-of-order issue superscalar processor with a 10 stage pipeline (fetch, decode, cluster assignment, rename, dispatch, issue, register read, execute, writeback and commit). The processor front-end (stages from fetch to dispatch), as well as the load/store queue and caches are centralized structures like in conventional superscalars, whereas the back-end features a partitioned model including distributed functional units, issue queues and register files.

2.1.1 Superscalar Model

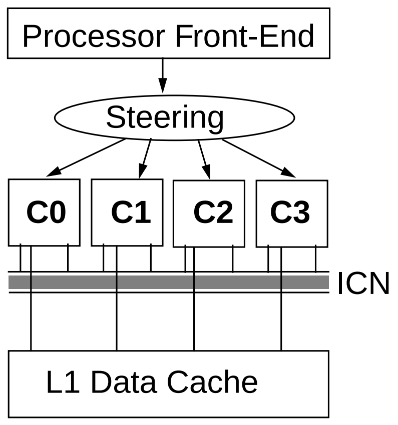
We assume a superscalar model similar to that of the MIPS R10000 [110], or the Alpha 21264 [42, 53], composed of separate integer and FP issue queues (IQ) for scheduling instructions, physical registers for storing both architectural and speculative state, a renaming mechanism that maps logical to physical registers, and a reorder buffer (ROB) for recording program order and holding instruction status, necessary to support speculation recovery and precise interrupts. Note that neither the ROB nor the IQ contain operand data but only physical register designators (tags), so in this model source operands are read after instruction issue and before execution, either from the register file or from the bypass. We assume an aggressive instruction fetch mechanism to stress the instruction issue and execution subsystems, and a minimum branch misprediction penalty of 10 cycles (7 pipeline stages between fetch and execute plus an additional 3-cycle delay for state recovery).

Loads and stores are divided into two operations, during the rename stage. One of them is dispatched to an integer issue queue and computes the effective address. The other one is dispatched to a load/store queue and accesses memory. When the first operation completes, it forwards the effective address to the corresponding entry in the load/store queue, to perform memory disambiguation and access to memory. A load access is issued when a memory port is available and all prior stores know their effective address. If the effective address of a load matches the address of a previous store, the store value is forwarded to the load. Store accesses are issued at commit time.

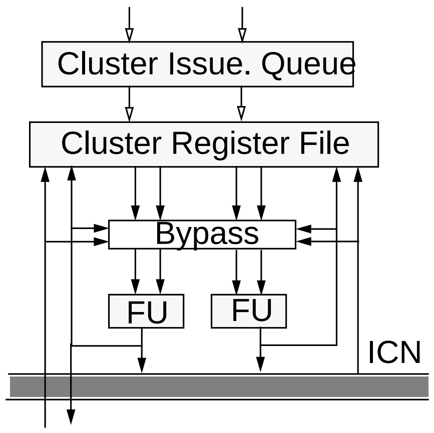
2.1.2 Clustered Model

The processor back-end is divided into N homogeneous clusters (see figure 2-1). Each cluster has both an integer and a floating-point datapath, each with its own instruction issue queue, a physical register file, a set of functional units, and the corresponding data bypasses among these functional units. Such a clustered organization also implies some extra activities in the front-end pipelines: instructions are assigned a cluster for execution (cluster assignment stage), then renamed (rename stage), and finally steered to the assigned cluster and written to the corresponding issue queue (dispatch stage).

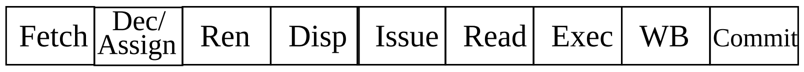
Local bypasses within a cluster are responsible for forwarding result values produced in the cluster to the inputs of the functional units in the same cluster. Inter-cluster bypasses are responsible for forwarding values among functional units of different clusters. Like in most existing superscalar architectures, the delay of a single ALU operation (e.g. addition) plus the delay of local bypasses within a cluster is kept into a single cycle to permit back-to-back execution of dependent instructions. Hence, local bypasses use short and fast wires that operate during the last cycle of execution. In contrast, inter-cluster bypasses require long and slow wires through the interconnection network, so they will take significantly longer [1]. Therefore, we assume a one-cycle latency for inter-cluster bypasses in the default configuration, although we also evaluate the effects of longer latencies. Latency is not the only penalty of inter-cluster communications. Bandwidth is also relevant, since it directly affects the number of register file



(a) Clustered processor back-end



(b) Detail of a cluster



(c) Pipeline stages

Figure 2-1: Reference Clustered Architecture

write ports, and the complexity of the wake-up and bypass logic. Our default cluster interconnect configuration assumes an unbounded amount of paths, in order to isolate our experiments from the effect of possible bandwidth bottlenecks, although we also evaluate the effects of having a limited inter-cluster communication bandwidth.

The register file is distributed among the clustered processing units [112, 29]. With a distributed register file model, unlike centralized or replicated models [52, 53, 70], a producer instruction initially writes its result only to the local register file of the cluster where it executes, so it only allocates a physical register in that cluster (there is a separate free-list per cluster). The distributed model requires less write ports to the register file than the other two models, and less physical registers than the replicated model. When a consuming instruction requires a source register produced in a different cluster, the register value must be forwarded from the producer cluster to the consumer.

In our model, all inter-cluster bypasses must be performed by explicit copy instructions. If instructions were allowed to read registers from remote clusters directly, the register files would be more complex (would have more read ports) and the read latency would be longer (send designator, read register and send value back). An alternative model could have avoided explicit copy instructions for registers that are unavailable when the consumer is dispatched, by forcing the producers to broadcast their result tags and values, but it would be at the expense of additional complex logic in the dispatch stage. Copy instructions may be generated either statically [29] or dynamically [73, 112]. In our model, to keep the clustered microarchitecture transparent to the ISA, copy instructions are generated on-demand and inserted dynamically by the rename logic. The copying process is briefly outlined below.

(a) Sample code: a copy is inserted to forward R1 from cluster 0 to cluster 2

Source code	**Cluster assignment**	**Renamed code**
`I1:   R1  = ...`	0	$P11_0$ = ...
		copy: $P15_2 = P11_0$
`I2:   ... = R1`	2	... = $P15_2$

(b) Mappings of R1 after renaming I1 and I2

	mapping0		mapping1		mapping2		mapping3	
R0								
R1	1	P11	0		1	P15	0	
R2								
		•••		•••		•••		•••
Rn								

Figure 2-2: Example of Map Table for 4 clusters

An instruction's logical destination register is initially renamed to a physical register in the cluster where it is going to be executed, using a separate free-list. The rename table keeps track of this mapping as well as the cluster that will produce the value (see an implementation example in figure 2-2). If a later consumer instruction is steered to a cluster where there is no physical copy of a source register, then the renaming logic allocates a physical register in the cluster of the consumer instruction and inserts a special copy instruction into the producer's cluster to forward the operand value. When the copy instruction is issued in the producer cluster, it reads the needed source register value, either from the bypass or from the register file, sends it through the cluster interconnection network, and writes it to the newly allocated physical register and local bypass network in the consuming instruction's cluster. After generating the copy instruction, the rename table is updated accordingly to show that the copied source register is now mapped to a physical register in the consuming instruction's cluster.

Since copy instructions do not override previous mappings, this mechanism may generate as many mappings per logical register as clusters. A further consuming instruction will use the mapping that corresponds to the cluster it is assigned to. Note that during the renaming of an instruction, just one physical register for its destination register is allocated. Additional physical registers to store copies of it in other clusters are only allocated on demand if they are required by subsequent instructions that do not execute in the same cluster. All these physical registers will be freed by the first subsequent instruction that writes to the same logical register, when it is committed. This scheme requires some degree of register replication which dynamically adapts to the program requirements and is much lower than replicating the whole register file. Compared with a full replication scheme, it has also less communication requirements and thus, less inter-cluster bypass paths and less register file write ports.

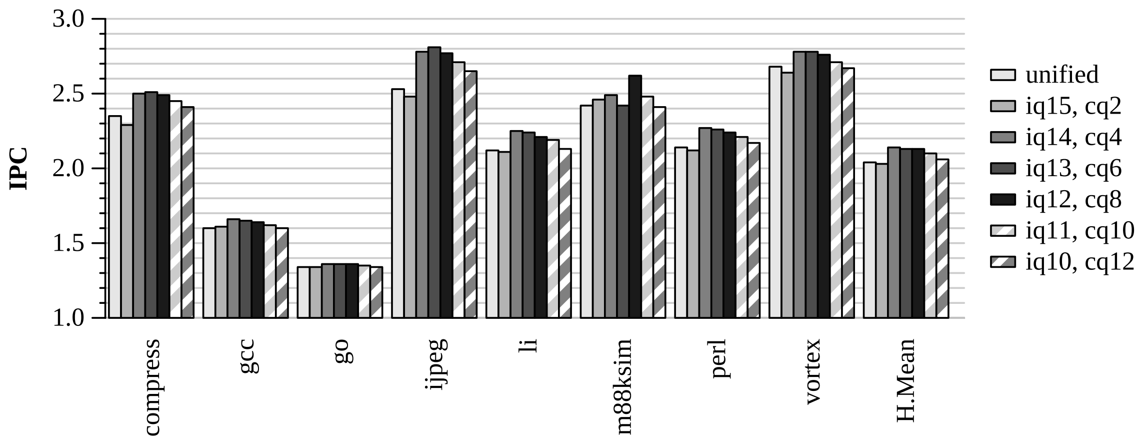


Figure 2-3: Performance improvements of splitting issue queues for copy/regular instructions, for various lengths of the issue queue and copy queue

2.1.3 Implementation Issues for Copy Instructions

If the processor has a limited number of inter-cluster bypass paths, they must be reserved by the issue mechanism like any other resource. Copy instructions provide a simple mechanism to allocate the required bypasses and schedule inter-cluster communications. They also provide a simple method for precise state recovery, since copy instructions are inserted in the reorder buffer like normal instructions. However, since a copy instruction makes the dependence chain one node longer, it increases by one cycle the total effective latency between the producer and the remote dependent instruction (in addition to the bus latency). A particular implementation could optimize this, either by shortening the tags propagation delay between clusters or by implementing specific hardware that avoids generating copy instructions.

Another optimization consists of using split issue queues for copies and regular instructions, thus preventing copy instructions from reducing the effective issue width and the effective length of the issue queues. This optimization could be implemented without any increase of the wake-up logic complexity and delay, which depends on the total length and loading of the tag broadcast wires. Actually, for a given wake-up delay, the total queue capacity of the split design may be higher than the unified one, because reservation stations for copy instructions only have to check one source register, so one regular reservation station may be substituted by two copy reservation stations. In other words, a unified design with queues of size S has similar wake-up delay as a split design with S_c entries for copies and S-$S_c/2$ entries for regular instructions. More importantly, the split design greatly simplifies the select logic since both queues apply for different kinds of resources. On the other hand, if copies are allowed to issue independently of regular instructions, one or more dedicated register file read ports must be added.

We developed some preliminary experiments with split issue queues that showed good performance improvements (see figure 2-3). Nevertheless, a thorough and accurate study of efficient issue mechanisms for copy instructions has been left for future work, and none of these optimizations is assumed in this thesis.

2.2 Main Architectural Parameters

We experiment with several cluster configurations (targeting different technologies and clock rates) with different complexities. Increased wire delays and clock speeds will demand clusters with more simple structures. We have chosen several possible scenarios with integer clusters issue widths of 8, 4 and 2 instructions per cycle. The integer register files have respectively 128, 80 and 56 physical registers per cluster, and the integer issue queues have respectively 64, 32 and 16 entries. K.Farkas [27] showed that performance tends to saturate around 80 physical registers (4-way/32-entry IQ), and around 128 registers (8-way/64-entry IQ).

Most experiments throughout this thesis consider an 8-way processor configured either as 4x2-way clusters, 2x4-way clusters, or a centralized conventional (1x8-way) back-end. The main default architectural parameters are described in table 2-1. Wherever other configurations are used, the differences are described in the text.

8-Way centralized front-end

Parameter	Common to all configurations
Fetch & decode width	8
L1 I-cache	32KB, direct mapped, 64-byte lines, 1-cycle hit time
Branch Predictor	Hybrid gshare/bimodal. Gshare has a 64K 2-bit counters PHT and a 16-bit global BHR. Bimodal has 2K 2-bit counters. The choice predictor has 1K entries of 2-bit counters
ROB	128 entries
LSQ	64 entries, loads may execute when prior store addresses are known
L1 D-cache	64KB, 2-way set-assoc., 64-byte lines, 2-cycle hit time, 3 R/W ports
L2 I/D-cache	256 KB, 4-way set associative, 64 byte lines, 6-cycle hit time
Memory	8 bytes bus bandwidth to main memory, 18 cycles first chunk, 2 cycles inter-chunk

Back-end (per cluster)

Parameter	8-Way (centralized)	4-Way	2-Way
Issue queue size (int/fp)	64/ 64	32/ 32	16/ 16
Issue width (int/fp)	8/ 4	4/ 2	2/ 1
Functional units (int/fp)	8(4 include mul/div)/ 4	4(2 include mul/div)/ 2	2(1 include mul/div)/ 1
Physical registers (int/fp)	128/ 128	80/ 80	56/ 56
Inter-cluster communic.	1 cycle latency		

Table 2-1: Default main architecture parameters

2.3 Simulation Methodology

The experiments were performed with a modified version of the sim-outorder simulator from the SimpleScalar tool set [15], version 3.0. This cycle-accurate execution-driven timing simulator was extended to include register renaming through a physical register file, issue queues (separate integer and FP queues), additional pipeline stages, a clustered back-end, and all of the microarchitectural details described in this section, as well as the techniques described in following chapters.

For the experiments in this thesis we have used the SpecInt95 [97] and Mediabench [56, 62] benchmark suites. All 21 Mediabench benchmark programs are run to completion, or up to a maximum of 300 million instructions (totalling 2,2 billion instructions). All SpecInt95 benchmarks are run to completion with the following inputs: *go* 9 9; *gcc* genrecog.i; *perl* scrabble.in; *m88ksim*, *compress* and *li* with the train inputs; *ijpeg* with a 88x31 pixel image, and *vortex* with the train database reduced to 1/10th. All the benchmarks were compiled for the Alpha AXP 21264 using Compaq’s C compiler with the -O4 optimization level.