CHAPTER 1

INTRODUCTION

This chapter presents the background and motivations behind this work, a brief description of related works, and an overview of the main proposals and contributions of this thesis.

1.1 Background and Motivation

Over the past decade superscalar microprocessors [50, 91] have achieved enormous improvements in computing power, and they form the basis for many computing systems ranging from desktop computers to massively-parallel systems. These processors achieve higher performance by executing multiple instructions in parallel every cycle, sometimes out of program order. The hardware is responsible for ensuring correct execution by monitoring instruction dependences and issuing them sequentially when appropriate.

However, in recent years several new architectural and technological constraints have arisen which will force superscalar architectures to evolve towards more decentralized designs, in order to preserve architectural long-term scalability. Many of the existing proposals are based on the concept of a clustered superscalar microarchitecture. This thesis focuses on one such clustered approach and proposes techniques to efficiently manage the most challenging problems that appear as a consequence of the partitioning of the main architectural structures.

1.1.1 The Performance of Superscalar Processors

In a typical superscalar architecture like the Alpha 21264 [42, 53] or the MIPS R10000[110], multiple instructions are fetched per cycle. When a branch is encountered, the program counter may be updated speculatively according to a branch predictor, to avoid stalling the fetch engine

until the branch is resolved. Instructions are then decoded, renamed and inserted into the reorder buffer (ROB, also referred to as instruction window), from where they exit when completed, in program order. Renamed instructions are also inserted into an issue queue. An instruction waits in this queue until its source operands are ready, either because they have been written to the physical register file, or because they will be available through the bypasses at the time the instruction executes. Multiple ready instructions may be chosen every cycle for execution by the issue logic, subject to the availability of execution resources. This technique for exploiting fine-grain parallelism is called instruction-level parallelism (ILP), and the maximum number of instructions processed in parallel is known as processor width.

The performance of a superscalar microarchitecture is inversely proportional to the execution time of programs, and for a given program or benchmark, it may be computed as the product of the average number of committed instructions per cycle (IPC) by the clock frequency. The IPC depends on a number of factors, including the inherent program ILP, the processor width, the size of the instruction window, and many other architectural factors. Clock frequency is the inverse of the clock cycle time, which depends on the delays associated with the critical paths of the architecture. Moreover, ignoring wire delays, cycle time may be computed as the product of the maximum number of logic levels in a single stage by the delay of each single logic gate. The delay of a single gate of logic depends on transistor switching time, and is reduced by shrinking transistor dimensions, which is usually achieved with each new fabrication process. The number of logic levels per stage depends on the amount of work done per stage, and may be reduced by increasing the pipeline depth. It is sometimes measured as the number of FO4 (fanout-of-four). A FO4 is the delay of an inverter driving four copies of itself, which is independent of process technologies [43].

In recent years, superscalar processor design faces significant challenges as higher levels of ILP are needed, and new technological constraints transform the old design rules. Among other issues, signals no longer propagate quickly across the chip, and power becomes a limiting factor to the high-performance computing segment. In the following sections these problems are briefly described and it is shown why clustered architectures may effectively deal with them.

1.1.2 Technology Scaling of Wire Delays

Ideally, if current processor designs were simply scaled down with successive technology shrinks, gate delays would also scale and performance would increase at the same pace. However, actual designs do not follow this model because wire delays do not scale at the same pace as gate delays. Let us first consider an idealized case, where all three dimensions in a circuit are shrunk by a factor of 1/s. To a first-order approximation, wire delay is proportional to $R_w*C_w*L^2$, where L is wire length and R_w and C_w are resistance and capacitance per unit length. Since L scales as 1/s, R_w scales as s^2 and C_w remains constant, wire delay remains roughly unchanged, in contrast with gate delay, which scales as 1/s [32, 102].

In practice [1, 43, 46], to partially mitigate resistance growth, the aspect ratio of wires (ratio of wire height to wire width) increases slowly through technologies, so that the height of wires scales down more slowly than feature size. At the same time, wire height causes an increase of the coupling capacitance component of C_w, although partially mitigated by reductions in the

dielectric constant of the insulator between the wires. In summary, while C_w is expected to increase only slightly, R_w will increase dramatically across technologies, so that wire delay per unit length will increase at a faster rate than wire length shrinks.

Actually, for wires that scale in length with technology, the magnitude of this problem is relatively small. The main problem is with global wires that do not scale in length, because they have an increasing logical span - they communicate across an increasing amount of gates - as technology scales. One possible solution to the wire delay problem is to insert repeaters at regular wire intervals, which allows wire delay to scale linearly with wire length, but this solution is not suitable to the dense wire packing of memory arrays without significantly increasing its area and, in addition, for the best case, it only allows to keep the wire delay constant [1], while gate delay decreases with each technology generation. Implementing such signals with wider and thicker wires, in middle or top metal layers is a common practice, but it is unlikely that the number of layers will increase fast enough to stay ahead of the problem.

1.1.3 Power Consumption

Energy dissipation has emerged as a primary design constraint for all microprocessors, from those in portable devices, where battery life is an issue, to those in high performance servers and mainframes, because of the limited heat-dissipation capabilities of packages. Dynamic power consumption in CMOS technology is proportional to the clock frequency, the transistor switching activity and capacitance and the square of the supply voltage. So, higher clock frequencies and transistor counts have made dynamic power a main concern in new processor designs [14]. For instance, the power consumption of a register file increases with its size, due to the higher switching capacitance, and the area grows linearly with the number of registers but quadratically with the number of ports. Since the number of ports is mainly proportional to the number of instructions that may access the register file simultaneously, the power consumption of a register file increases as the square of the processor width.

Most of the power reduction in high-performance processors was achieved traditionally through supply voltage reduction and process shrinks. Maintaining high transistor switching speeds, however, requires to reduce transistor threshold voltage, giving rise to a significant increase of leakage energy dissipation even when the transistor is not switching. The resulting static power consumption will likely become dominant in the forthcoming technology generations.

1.1.4 Clock Speed

Over the past decades, the clock period of microprocessors has experienced a dramatic reduction. For instance, Intel data from six x86 processor generations shows a reduction in the number of FO4 delays per pipeline stage from 53 in 1992 (i486DX2) to around 12 in 2002 (Pentium4), corresponding to substantially deeper pipelines [1, 49]. Many works have shown that arbitrarily reducing the amount of work per pipeline stage - hence increasing pipeline depth - does not necessarily lead to higher performance, because of the overheads due to latches between stages, clock skew and jitter. However, these works also show performance curves with optimal points at clock periods below those implemented in current microprocessors.

Hrishikesh et al. estimated the optimal logic depth per pipeline stage to be between 6 to 8 FO4 delays, depending on program characteristics [44], and E.Sprangle and D.Carmean [95] concluded that further performance improvements are still expected by increasing pipeline depth beyond that of the Pentium4. However, reducing the clock period and increasing pipeline depth also requires using more complex structures in hardware: bigger storage structures (such as the reorder buffer or the physical registers), more levels of bypassing among stages, new prediction mechanisms (such as the load hit/miss predictor to help scheduling load-use instructions [42, 53]), etc.

More recently, Srinivasan et al. [96] reported that 10 FO4 delays is the optimal clock period if the pipeline is optimized for performance. However, they also found that optimizing for energy-efficiency gives an optimum of 18 FO4 delays. This result indicates that power constraints may force superscalar designs to target lower clock speeds, and therefore lower total performance. Thus, as long as power constraints dominate, achieving more performance by stretching the pipeline length will require major changes in the architecture that help improve its power efficiency.

1.1.5 Complexity

The attainable IPC rate of a superscalar processor increases with processor width, and also with the size of the instruction window [40], the register file, the caches, the predictors, etc. The quest for exploiting higher levels of parallelism in high-performance architectures, fueled by the exponential increase in transistor budgets, has brought in the past decade growing levels of complexity to these processor components. These trends are likely to continue in the near future, where wide superscalar processors with multithreading capabilities will provide support to software with many parallel, independent transactions, such as web servers, and other general-purpose applications.

The complexity of a design may be variously quantified in terms such as the number of transistors, die area, and power dissipated. In this thesis we borrow the definition of S. Palacharla, who defines the complexity as the delay through the critical path of a piece of logic [70, 71]. The complexity depends on factors such as the number of logic stages, the length of wires, the degree of fan-out of a particular signal, etc. The complexity of most architectural structures grows with their size, because of the longer wires and deeper levels of logic required to implement. Palacharla, Jouppi and Smith [69, 70, 71] developed simple analytical models to estimate the delay (complexity) of the issue logic, bypass logic, rename logic and register file. They showed that the delay of the issue logic scales quadratically with the issue width and window size, and the delay of the bypass logic scales linearly with issue width. In addition, they showed that the wire component is a significant fraction of the total delay and, because wire delays scale slower than gate delays, it will tend to dominate the total delay in future technologies, thus making these structures to scale even worse.

Palacharla, Jouppi and Smith concluded that both the issue logic and the bypass will likely become the most critical structures as architectures move towards wider-issue, larger windows and advanced technologies in which wire delays dominate. The expected overall performance

benefits of wider architectures may be offset by the higher complexity of its components, because they may require longer critical paths in the architecture and a slower clock.

1.1.6 The Clustered Approach

Many previous works (see [1, 14, 43, 61, 92] among others) have made evident that the old superscalar paradigm, with large monolithic structures will not be suitable to face the problems arisen by the above technology and architectural trends: the growing size of many architectural components implies superlinear increases in power consumption and complexity, which are further exacerbated by decreasing clock periods and the poor scaling of wire delays.

As a consequence of these trends, in a conventional superscalar architecture the access to many processor structures such as the issue buffer, the result bypass network, the register file, the rename table, etc. will require in the future multiple clock cycles. Even though these structures were perfectly pipelined, some of them stay in tight hardware loops with frequent recurrences, so that lengthening the loop latency by one cycle implies inserting many bubbles into the pipeline, with a substantial negative impact on performance.

The more paradigmatic example of this is the issue logic. The issue logic in a superscalar processor must wake up a number of instructions when their operands become ready, and select some of them for execution. Should it take more than one cycle, then the processor would be unable to issue dependent instructions in consecutive cycles [70, 98]. A similar problem occurs with result bypassing. Most simple integer instructions are executed in a single cycle. After the ALU has computed the result, it must still be routed through the bypass network to the appropriate input multiplexers within the same cycle, in order to be used by dependent instructions. Taking an additional cycle for bypassing would prevent to execute dependent instructions in consecutive cycles. As a further example, though less critical, increasing the number of cycles to access the register file degrades performance since it increases the number of bypass levels, increases the branch misprediction penalty, and increases the cost for misscheduled dependent instructions after a load miss [13, 21, 53]. Other critical hardware loops occur in the instruction fetch address generation and the memory disambiguation. It is therefore of paramount importance that all these processing tasks are executed as fast as possible.

Clustered microarchitectures are a recently proposed architectural paradigm that addresses the above problems. Part of the architecture is partitioned into smaller and simpler units, called clusters, running at a high clock rate. Critical processor structures are partitioned among the clusters, so they are still accessed quickly. Many communication (wire) delays among clusters are made explicit to the microarchitecture, so they may be handled more efficiently to minimize their impact on performance: critical communications are localized as much as possible within the small clusters, where they are fast, whereas other less critical communications go through global inter-cluster paths that may take relatively long to propagate. Compared to a conventional centralized architecture, the IPC of the clustered architecture may be lower because of the extra communication latency and a less flexible resource usage. However, performance is the product of IPC by clock frequency. So, when cycle time is factored in, the clustered architecture outperforms the centralized one because it may be clocked faster [70]. In addition, because of

wire delays, this advantage tends to increase with larger processor widths and smaller feature sizes.

Also, in a clustered architecture, the size reduction of its structures means a substantial reduction in power consumption and a better power-efficiency. For instance, for a similar clustered architecture to the one we study in this thesis, V. Zyuban showed that its energy-delay product is lower than that of a comparable centralized architecture, and that this advantage increases with larger issue widths [112, 113]. This means that for a given power budget, a clustered architecture achieves a higher IPC than a conventional superscalar. Furthermore, the partitioning of structures may simplify power management via techniques such as selective clock/power gating [7] and voltage scaling.

The design of a clustered microarchitecture is currently an active research area, with many proposals pushing ahead the state-of-the-art in multiple directions. The goal of this thesis is to contribute to these trends with new architectural improvements on key aspects of a clustered microarchitecture. Our research focuses on cluster assignment algorithms, techniques for reducing inter-cluster communications, the design of the cluster interconnect and the design of a clustered front-end. All of these proposals apply to a typical clustered superscalar architecture that features a fully distributed register file, issue window and bypass network. Before describing them in more detail, we first review in the next section the most relevant related work.

1.2 Related Work

Many different architectural paradigms have been proposed for structuring a processor into clusters, at several degrees of granularity. This section outlines the existing work related to clustered superscalar processors. Note that, though not covered in our short review, clustering approaches are also common in VLIW architectures [9, 19, 20, 22, 26]. The design space in clustered VLIW architectures is well researched [67, 84], and many processors in the DSP/ embedded domain use a clustered microarchitecture, such as the Texas Instruments TMS320C6000 [101] or the Analog's TigerSharc [35], for instance.

Clustered superscalar microarchitectures partition execution resources into different processing units, and instructions are steered to either unit according to some kind of cluster assignment scheme. N. Ranganathan and M. Franklin [79] describe a taxonomy of decentralized ILP execution models. They classify clustered approaches by their instruction partitioning, which may be based on resource dependences, control dependences or data dependences, although combined approaches are possible as well. The first class has been implemented since early dynamically scheduled processors such as the Tomasulo-based IBM 360/91 [103], and it groups instructions that use the same execution units, i.e. by instruction types, such as integer, floating point, memory, branches, etc. Other examples of these are the MIPS R10000 [110], the Alpha 21264 [42, 53], the PowerPC 604 [94], etc. The second class groups program segments made of consecutive control-dependent instructions (called tasks, threads or traces), and assigns them to a different cluster or processing element (PE) for parallel execution. Their main emphasis is on increasing parallelism by means of control speculation - but also data and

memory speculation - so they are often classified as speculative multithreaded architectures. Examples of these are the Multiscalar [32, 34, 93], Trace Processors [82, 83], and several other speculative multithreaded architectures like the SPSM [24], Superthreaded [106], Dynamic Multithreading [6], Speculative Multithreading [59, 60], etc. This thesis focuses mainly on a third class of clustered superscalar architectures whose cluster assignment scheme groups every chain of data-dependent instructions into the same cluster, in order to reduce the number of communications between clusters. Some of these architectures leverage some concepts from the previous Multiscalar project, but make the emphasis on reducing hardware complexity and/or power consumption. Some of them, such as the RAW architecture [109], the Grid Processors [49, 63, 85], the Multicluster [28, 29], the Integer-Decoupled [68, 71, 87], or the ILDP [54, 92], assign the instructions to clusters at compile time, while others, such as the Dependence-Based [70, 71], PEWs [3, 52, 79], the Energy-Efficient Multicluster [112, 113], the Alpha 21264 processor [53], or the schemes proposed in this thesis [73, 74, 75] perform dynamic cluster assignment. Below we describe in more detail the works that are more closely related to this thesis.

1.2.1 The Two-Cluster Alpha 21264

The Alpha 21264 processor features a clustered integer execution core with two four-way issue clusters. The register file is replicated in each cluster, and both copies are kept consistent by broadcasting the results of all computations to both files, so no explicit transfer instructions are required. The values produced in one cluster are made available to the other cluster one cycle later, because of inter-cluster wire delays. Compared to a centralized architecture, this clustered scheme requires half the number of read ports in each register file, reduces the complexity of the local bypass network and enables a faster clock. However, compared to a distributed register file scheme, the complexity, area and power consumption of the replicated register file is higher because it requires more registers, write ports, and register file activity. Cluster assignment is done at issue time and is handled by the centralized issue logic itself with a simple and effective algorithm: each instruction is steered to the cluster that has first available its source registers and execution resources. The centralized issue queue does not help reducing wakeup complexity but it allows delaying the cluster assignment until issue time, which is most effective to avoid inter-cluster communication penalties and workload imbalance. To compensate for wakeup logic complexity, the selection logic is greatly simplified by pre-assigning (“slotting”) instructions at decode time to one of the two possible functional units in a cluster.

1.2.2 The Integer-Decoupled Architecture

S. Palacharla and J. E. Smith [67, 71] proposed the Integer-Decoupled architecture. They observed that most conventional superscalar processors, such as the MIPS R10000 or the Alpha 21264, actually include two clusters for integer and floating point instructions, with separate datapaths, register files and issue queues. However, when these processors execute integer codes, most of the floating point resources are idle. The proposed Integer-Decoupled clustered architecture is a cost-effective approach that exploits these resources by adding some simple integer ALUs to the floating point cluster, which increases the effective integer issue width with minimal hardware cost. This architecture has no support for copying between the register files, and it supports load/store instructions only in the integer cluster. Thus, the cluster assignment

scheme is greatly constrained to steer all loads/stores to the integer cluster and to avoid all cross-cluster register dependences except those already supported on most current superscalars: between a load and its dependent use instructions, or between a store and its data operand producer.

In that work, they did not evaluate performance but proposed a cluster assignment algorithm that satisfies the above constraints, and used it to evaluate the amount of integer workload that could be off-loaded to the floating-point cluster. These constraints are similar to those found on access/execute decoupled architectures [90, 72, 104], so the partitioning bears many similarities. Their cluster assignment algorithm works in two phases. First it walks all the dynamic instruction stream to identify non-disjoint program partitions referred to as slices: the load/store slice, which includes all loads and stores, and instructions involved in address computations; the branch slice, which includes all conditional branches and instructions involved in the computation of branch conditions; and the compute slice, which includes the rest of instructions. The second phase assigns each static instruction to a cluster in the following way: first, the load/store slice is entirely assigned to the integer cluster; second, the instructions in the branch and compute slices that are not dependent on instructions in the load/store slice are assigned to the floating-point cluster; and finally, the rest of instructions are assigned to the integer cluster. The authors found that between 10% to 39% of the instructions in integer codes may be executed in the floating-point cluster, and that complex integer instructions were never assigned to the floating-point cluster, so there is no need to duplicate such gate-intensive execution units.

S.S. Sastry, S. Palacharla and J.E. Smith further developed a pure static cluster assignment scheme for the Integer-Decoupled architecture [87], with the goal of off-loading as many instructions as possible to the floating-point subsystem. The initial architectural constraints are relaxed by adding support for copying registers between clusters, both in the ISA and in the microarchitecture. The cluster assignment scheme is an enhancement of the above Palacharla's load/store slice partitioning [67]. It is extended to allow generating copy instructions as well as duplicating certain instructions in both clusters, based on a heuristic cost model that estimates the benefits and overheads of each possible copy/duplication.

Some of the cluster assignment schemes we present in this thesis [17] are tested on a specific superscalar microarchitecture inspired in the above Integer-Decoupled approach. Our work differs in that we initially propose a dynamic version of the "load/store slice" cluster assignment, and then we propose further effective improvements as well as different schemes not based on slices. Other differences are 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.

1.2.3 The Dependence-Based and Architectures with Replicated Register Files

S. Palacharla, N. P. Jouppi and J. E. Smith proposed the Dependence-Based microarchitecture [69, 70, 71]. Its primary motivation is to simplify the issue logic to allow a faster clock by replacing the conventional associative issue buffer with a set of FIFO queues, each containing

a chain of dependent instructions. The dispatch logic follows the strict rule of appending an instruction to a non-empty issue queue only if it directly depends on the last instruction in that queue, otherwise it must be steered to an empty queue. This rule ensures that ready instructions always reside at the head of their queues, so the issue logic has to monitor just the head of these queues. It is also proposed a clustered version of the Dependence-Based architecture where execution resources are further partitioned into symmetrical clusters to simplify the register file and the bypass network. As with the Alpha 21264, registers are replicated in all clusters, but the FIFO issue queues are distributed among the clusters and instructions are steered during dispatch. The cluster assignment algorithm begins by steering instructions to one cluster and it keeps steering instructions to the same cluster (following the above dependence-based dispatch rule to choose a queue) until it needs a new empty queue and there is not one in that cluster. Then, it switches to the "next" cluster. This heuristic lacks of any explicit mechanism to balance the workload, which is somehow adjusted implicitly by the distribution of instructions to issue queues.

H-S. Kim and J.E. Smith proposed the Instruction Level Distributed Processor [54], which borrows the issue mechanism from the Dependence-Based paradigm: it steers chains of data dependent instructions (strands) to FIFO issue queues. However the ILDP paradigm defines a new accumulator-based ISA which uses global registers for values passed among different strands, and accumulators for passing local values among instructions within a strand. The task of partitioning the program code into strands is left to the compiler (or binary translator), and communicated to the microarchitecture through the assigned accumulator names. The static partitioning algorithm assigns global general-purpose registers to values that have a relatively long lifetime and are used many times, whereas it assigns an accumulator to values that are used only once or a small number of times in close proximity.

The steering algorithm proposed by S. Palacharla, N. P. Jouppi and J. E. Smith for the Dependence-Based architecture was also evaluated for a clustered architecture having associative issue windows instead of FIFOs [70]. The steering algorithm does not require any modification except that it sees the issue window of each cluster modelled as a collection of FIFOs with instructions capable of issuing from any slot within each individual FIFO. They found almost no IPC difference between the two clustered approaches.

A. Baniasadis and A. Moshovos assumed a similar clustered model, with associative issue windows and replicated register files, to analyze various adaptive and non-adaptive cluster assignment schemes, for four clusters [10]. Their non-adaptive schemes assign sequences of consecutive instructions to clusters, where clusters are alternatively chosen in round-robin order. These algorithms differ in the criteria used for delimiting the sequences: the First-fit ends a sequence when the cluster window fills-up; the Modulo schemes (MOD_n) define fixed-length sequences of n instructions; the Branch-cut ends sequences at branch boundaries, and the Load-cut ends sequences at load boundaries. These schemes do not handle communication and workload balancing explicitly. These algorithms were compared to SLC, a variation of our "Ldst Slice Balance" algorithm [17], and to DEP, a dependence-based scheme similar to our "Priority RMB" algorithm [73] (also described in chapter 4). They found MOD_3 to be the best performing scheme, closely followed by SLC (within a 4%). However, other works [3], and our studies as well (see chapter 4), have observed worse performance for modulo-based than for

dependence-based algorithms. Their adaptive algorithms are variations of the above schemes by applying two strategies. The CNT adaptive methods record, for every instruction, the success or failure of past assignment history, consisting on a 2-bit saturating score per cluster. The score decrements if the instruction suffers an issue delay after it becomes ready, or increments otherwise, so that future instances of the same instruction are assigned the cluster with the highest score. The MOD_a is an adaptive version of the MOD_n scheme, where n varies periodically in search of the best performance. Issue stall statistics are collected during a period of time, after which the length n is increased or decreased. If an increase/decrease produced an improvement with respect to the previous period of time, then the length is again increased/ decreased, otherwise the sense of the variation is reversed. They found that the best schemes were CNT-SLC and MOD_a.

E. Tune, D. Liang, D.M. Tullsen, and B. Calder [107] also assumed a similar clustered model, with associative issue windows and replicated register files, for two and four clusters. They proposed a critical-path predictor and proposed modifications to two dependence-based cluster assignment schemes to take into account instruction criticality. These baseline cluster assignment schemes (Reg and Act_reg) were similar to our Advanced RMB and Priority RMB schemes [16, 73] (also described in chapter 4). In their proposed schemes, critical path prediction is used to break ties when an instruction has two operands produced in different clusters, then it is assigned to the cluster of its critical predecessor. C. Fields, Rubin, and Bodik [31] conducted similar studies with the same cluster assignment scheme also modified to break ties using criticality information, but with a more effective token-passing critical-path predictor they proposed.

The PEWs (parallel execution windows) microarchitecture [52] follows similar motivations for reducing complexity and improving scalability of the issue logic. It partitions the datapath and issue buffer into multiple symmetrical clusters while keeping a centralized, versioning-based register file. Source registers are read during decode if available, otherwise they are read later from the local bypass or from the unidirectional ring cluster interconnect. To avoid the huge traffic overhead that produces the broadcast of all result values through the interconnect, and also to avoid using explicit transfer instructions, the authors assumed some kind of mechanism in the decode stage that “informs the relevant cluster which results must be forwarded”, but unfortunately it is not described. In the PEWs architecture, the steering scheme is quite simple: it assigns an instruction to the cluster where the source operand is to be produced, except if it has two operands that are to be produced in different clusters, in which case the algorithm tries to minimize the forwarding distance of the operands. This algorithm also lacks of a workload balance mechanism. Aggarwal and Franklin [3] studied the scalability of several previously proposed instruction steering algorithms on a PEWs architecture when increasing the number of clusters, both for ring and crossbar interconnects. However, since their study scaled simultaneously various parameters like the issue width, communication latency and number of clusters, it is difficult to isolate their individual effects and draw conclusions. Overall, they found that algorithms that work well for four or fewer clusters do not scale well to more than four clusters.

For architectures with replicated register files and a dynamic cluster assignment, such as those reviewed above [10, 31, 52, 70, 107], the cluster assignment scheme does not need to

group two dependent instructions in the same cluster if the value of the producer is already available at the time the consumer is assigned a cluster, hence they are not suitable for distributed register files like the one considered in this thesis. In addition, these cluster assignment schemes lack an explicit mechanism to address the load balancing problem.

1.2.4 Cluster Assignment Schemes Based on a Trace Cache

Several works have explored cluster assignment schemes that exploit the opportunities offered by trace caches. A trace cache [49, 63, 76, 81] can store not only dynamic decode and renaming but also data dependences and cluster assignment information.

N. Ranganathan and M. Franklin [79] proposed extending the PEWs paradigm with a trace cache and a register dependence preprocessing unit (RDFG) that analyzes the trace once and stores this information in it for future reuse. Thus, this complex task is effectively off-loaded from the critical path, and cluster assignment logic gets simplified. The cluster assignment scheme considers both intra-trace and inter-trace dependences: traces are first divided into chains of data-dependent instructions, and instructions at the head of each chain are assigned to clusters according to data dependences with previous traces, as in the first PEWs scheme; then, the rest of the chain is assigned to the same cluster as its leader.

Other approaches propose to not only off-loading the data-flow analysis from the critical path but also the cluster assignment task itself, by performing it in the fill unit, during retirement. D.H. Friendly, S.J. Patel and Y.N. Patt [36] proposed that the fill unit reorders instructions within a trace, based only on intra-trace data dependences. More recently, R.Bhargava and L.K. John [11] proposed a mechanism to take into account also inter-trace dependences. Unlike intra-trace dependences, inter-trace dependences are dynamic by nature, and are subject to change from one execution to another. Fortunately, critical inter-trace dependences come from the same static producer instruction 90% of the time. The proposed scheme records previous inter-trace dependences and their cluster assignments at retirement time, thus acting as a profiling mechanism capable of predicting future behavior.

1.2.5 The Multicluster and Architectures with Distributed Register Files

By distributing the register file among the clusters, an architecture may achieve lower complexity and power consumption than one with a replicated one. With a distributed register file, each result is typically written only to the register file of the producer cluster, thus a single physical register is allocated to each result, and each instruction has direct read access only to its local register file. Compared to a replicated register file organization, this technique reduces the number of write ports and the total amount of physical registers required. The architecture provides mechanisms to transfer register values from one cluster to another when the producer and the consumer cannot execute in the same cluster. The inter-cluster communication rate is obviously lower than for replicated register files because, instead of broadcasting all results to all clusters, only some results must be sent from one cluster to another. As long as the cluster assignment succeeds at keeping a low communication rate between clusters, a cost-effective architecture with distributed register files may constrain the cluster interconnect bandwidth

without any significant performance loss, to reduce complexity at several places: less comparators per issue queue entry are required because less result tags are broadcast to issue queues of other clusters; less bypass paths are required in the cluster interconnect, etc.

K.I. Farkas et al. proposed the Multicluster architecture [29, 28]. This architecture partitions datapaths, functional units, issue logic and register files into two symmetrical clusters. Some architectural registers which are defined global, are replicated in both clusters and must be kept consistent by replicating the instructions that produce them. The rest of architectural registers are assigned to either cluster based on their even/odd names, so that the register files are kept simple. Whenever an instruction has not all of its source and destination registers in the same cluster, it is split ("dual-dispatched") at dispatch time into two (or three) instructions: the "master" instruction performs the actual computation, while the "slave" (or slaves) perform a simple register transfer (either of a source or a destination register) through the appropriate transfer buffers. The dispatch hardware follows quite strict rules to distribute instructions to either cluster based on their register names, with almost no possibility to compensate for run-time conditions. The major task in distributing instructions is done by the compiler, which is responsible for assigning source and destination logical registers in a way that minimize communication and workload imbalance, based on estimations. K.I. Farkas also proposed a dynamic scheme [28] that adjusts run-time excess workload by re-mapping logical registers. However, he found most heuristics to be little effective since the re-mapping introduces communication overheads that offset almost any balance benefit.

V.V. Zyuban and P. Kogge proposed a version of the Multicluster architecture [112, 113] which is mainly concerned with the energy growth problem. They determined by simulation the optimal number of clusters and configurations to minimize the energy-delay metric, and showed that this architecture is more energy-efficient than centralized ones. The Energy-Efficient Multicluster architecture is quite similar to the one we study in this thesis in many aspects. Register files are distributed among clusters so that each cluster contains a dynamic subset of the physical registers, and all subsets are disjoint. Each cluster is provided with a local issue window, local register file, a set of execution units, local disambiguation unit, and one bank of an interleaved data cache. Most of the proposals presented in this thesis [16, 18, 73, 75, 74] assume a specific clustered microarchitecture (see chapter 2) similar to the above model. However, our inter-cluster register communication mechanism differs from that of the Zyuban's Multicluster in two main ways: first, values that communicate among clusters in our model are written in the destination cluster register file instead of using specific Remote Access Buffers, so that some physical registers get replicated in several clusters; and second, inter-cluster copy operations in our model sit in the issue queues, instead of using special Remote Acces Windows. None of these differences by themselves should have an impact on the IPC, but our assumptions are more constraining for two reasons: first, we assume that copies consume issue bandwidth whereas they do not; and second, we assume that copies require one clock cycle for issue plus additional transmission cycles, whereas they assume only a single cycle for issue and transmission.

The cluster assignment logic proposed for the Energy-Efficient Multicluster architecture assigns instructions to clusters based on register dependence and cluster workload information determined by the number of free entries in the issue windows. This scheme evaluates a cost

function for every possible assignment of each instruction, and then it chooses the cluster with minimal cost. The cost function sums the number of inter-cluster dependences minus the number of free entries in the corresponding issue window, multiplied by appropriate weights (unit weights were assumed for most experiments [112]). This quite simple scheme was intended to avoid hardware complexity, but it sacrifices too much effectiveness. Instead, this thesis proposes several more effective - though slightly more complex - algorithms (see chapter 4), whereas the cluster assignment logic complexity is addressed through clustering the front-end [74] (see chapter 7).

A. Seznec, E. Toullec, and O. Rochecouste proposed the WSRS architecture, which adopts a hybrid design point between a replicated and a distributed register file [89]. It attempts to achieve most of the flexibility of a replicated register file while keeping the complexity close to that of a distributed one. In the WSRS, every register file has its read/write ports connected to a pair of adjacent clusters. The physical registers are grouped into four disjoint subsets, each dedicated to the results produced in one cluster. The subset associated to a cluster is kept replicated in its two adjacent register files, so every register file holds copies of the two subsets associated to its adjacent clusters. Each cluster has read access to the three subsets held in its two adjacent register files, but read accesses to its left/right register files are further constrained to left/right source operands respectively. In this way, for any possible mapping of two source registers there exists at least one cluster where they are both accessible. The cluster assignment algorithm determines this cluster, and when there is more than one option, it breaks ties randomly. To increase the degrees of freedom of the cluster assignment, source operands of commutative operations may be exchanged prior to dispatch, and the hardware that executes non-commutative operations is adapted to admit operands in the reverse order. The authors observed that their cluster assignment algorithm produces high workload imbalances, so they suggested that including balancing considerations could benefit performance.

1.2.6 Other Related Works

Regarding communication reduction techniques, A. Aggarwal and M. Franklin explored several schemes to generate dynamically instruction replicas to reduce inter-cluster communications [4], either communications among the set of instructions being dispatched, or communications predicted with a simple history table that records the destination of every value forwarded in the past. Their replication schemes proved mostly effective for cluster assignment schemes that generate many communications (such as MOD_3), and for architectures with generous resources (16-way issue, 32-entry window per cluster, replicated register file, and a fully connected crossbar). Instruction replication had been proposed previously with other purposes: K.I. Farkas, N.P. Jouppi and P. Chow proposed dual-dispatching of instructions that produce global values [29] for a Multicluster architecture, to keep these values replicated; and S.S. Sastry, S. Palacharla, and J.E. Smith proposed instruction replication in their static scheduling algorithm for an Integer-Decoupled architecture [87], to help off-loading the integer cluster. Our approach to reduce inter-cluster communication is different, because we propose to predict the values of source operands produced in another cluster. Value prediction has been extensively explored in the past. Just refer, for instance, to the works of Lipasti et al. [58] and Sazeides et al. [88] that address the limits of true data dependences on ILP, and propose exposing more ILP by

predicting addresses and register values. Also, predicting the values that flow between different threads (in different clusters) is present in many speculative multithreaded architectures, such as the Trace Processors [82].

Regarding the design of interconnects for clustered superscalar architectures, A. Aggarwal and M. Franklin evaluated a crossbar and a ring for a PEWs microarchitecture [3], although their study focuses more on the scalability of the steering algorithms rather than on the design of the interconnects themselves. They further extended the study with a hierarchical interconnect for a large number (8 to 12) of 2-way issue clusters [4]. The topology they propose is a ring of crossbars that connects a small number of physically close clusters using a low-latency crossbar, and the distant clusters are connected using a ring. More recently, K. Sankaralingam et al. [86] describe a taxonomy of inter-ALU networks which includes, among others, conventional broadcast schemes as well as multi-hop interconnects like the one we proposed [75]. Through detailed circuit analysis at 100nm technology, they estimate communication delays for single-hop and multi-hop interconnects. They show that the latter ones scale much better than broadcast networks, which suffer primarily from wire delays resulting from significantly larger area required for wiring. The performance of several simple point-to-point and broadcast interconnects are evaluated for a conventional VLIW architecture and a Grid Processor architecture [64] having issue widths between 4 and 16, and it is shown that operand broadcast is not necessary in these architectures. Interconnect models for clustered VLIW architectures are also analyzed in a recent work [100].

Related to the partitioning of the processor front-end, P.S. Oberoi and G.S. Sohi proposed to parallelize the instruction fetch and rename tasks in the front-end [66]. They proposed a trace cache-like fetch unit that walks through the program stream by predicting traces [48] rather than individual branches. However, instead of fetching sequentially from the I-cache when the trace cache misses, they propose to assign predicted traces to different sequencers that can fetch instructions in parallel. Each sequencer stores its trace into a trace buffer entry instead of the fill-unit doing it. Although multiple buffer entries are written in parallel, they are allocated in order, so the sequential program stream is rebuilt in the buffer and traces may be consumed in order. Moreover, since buffer entries are not erased until their space is required, a predicted trace can be reused if it still stays in the buffer, so the fetch buffer acts like a small trace cache. Compared to a trace cache, this mechanism can exploit only a fraction of the locality, but it features a more powerful parallel fill mechanism. This mechanism improves net throughput over a sequential one with the same instruction cache bandwidth because it can reorder cache accesses to accommodate cache constraints like misses or bank conflicts. However, although the assumed instruction cache is highly banked, parallel access requires a complex arbitration and crossbar interconnect. In addition, the trace predictor is assumed to deliver one prediction per cycle, but it is still a complex centralized structure. Our work differs from this because our approach to partitioning the predictor and the fetch unit focuses on reducing complexity of the predictor and the instruction cache, rather than increasing throughput.

P.S. Oberoi and G.S. Sohi also found that, after a trace misprediction, the serial renaming task limits the final throughput to that of a single sequencer. Therefore, they proposed to distribute the conventional renamer into multiple trace renamers and allow them to rename in parallel from different partially fetched traces, by using a live-out predictor to speculate on inter-

trace dependences. Using the live-out prediction, a speculative version of the map table is created whenever the first instruction of a trace begins renaming, and it is sent to the next trace renamer, to allow it to begin renaming in parallel, speculatively. Distributing the renamer increases parallelism and performance for a front-end with parallel fetch, but has a negative IPC effect on a conventional machine that fetches traces sequentially. Our approach differs from this because we focus on distributing a serial renamer instead of a parallel one, as well as the cluster assignment logic, with the goal of reducing complexity. We specifically address the latencies produced when exchanging information between partitions.

1.3 Thesis Overview and Contributions

The goal of this thesis is to propose novel and effective techniques that address some of the critical design parameters of a clustered microarchitecture, with the objective of improving performance while keeping complexity low. Our main contributions are cluster assignment algorithms, techniques for reducing inter-cluster communication, the design of the cluster interconnect and the design of a clustered front-end. All of these proposals apply to a typical clustered superscalar architecture with data dependence based code partitioning that features a fully distributed register file, issue window and bypass network. The following sections outline the problems we are trying to solve, the approach we take to solve the problem, and the novel contributions of our work.

1.3.1 Cluster Assignment Algorithms

Probably the most critical design issue in a clustered superscalar architecture are the penalties of inter-cluster communications and workload imbalance. Inter-cluster communication occurs between dependent instructions that are assigned to different clusters, and it prevents them from executing in consecutive cycles due to the latency of the communication. Workload imbalance appears as a consequence of the partitioning, since having local resources prevents an instruction in one cluster from using a resource that sits in another cluster. Under limited resources, it may happen that instructions in an overloaded cluster are stalled due to the lack of local resources, even though idle resources exist in other clusters. Both workload imbalance and inter-cluster communication penalties negatively impact performance if they affect to critical instructions, and both issues are mostly influenced by the cluster assignment algorithm. The goal of our first research contribution is to propose better dynamic cluster assignment schemes for distributed register files that minimize the penalties of inter-cluster communications and workload imbalance.

Firstly, a set of algorithms are proposed that make cluster assignments based on groups of dependent instructions called slices [17]. The intuition behind this approach considers that two of the processor events that most hurt performance are cache misses and branch mispredictions, so it is likely that load address and branch condition computations stay in the critical path of execution of programs. The subset of instructions involved in one such computation is referred to as a slice. It seems a good program partitioning criterion to keep all the instructions of a slice within the same cluster to avoid incurring communication penalties in its execution. Our first

proposal is a dynamic steering scheme (*LdSt slice steering*) which is largely inspired on the early feasibility studies of Palacharla [68] and the static implementation of Sastry [87]. We first show that this simple dynamic approach outperforms the static one, but it is also shown that it does a poor workload balancing (as it was also recognized in the static proposal [87]). Therefore, our main contributions in that respect are several new dynamic schemes that evolve from the *LdSt slice steering* by improving the workload balance in several ways. The evaluation shows that our best slice-based scheme (*priority slice balance*) outperforms a state-of-the-art previous dynamic proposal [70], because it generates a significantly lower number of inter-cluster communications.

Secondly, a set of algorithms are proposed, which are referred to as Register Mapping Based (RMB) schemes, that make the cluster assignment of every instruction according to its direct dependences with its immediate ancestors [16, 18], rather than grouping chains of dependent instructions and assigning them as a group. Hence, a RMB scheme decides at a finer granularity, which allows it to be more flexible to keep the workload balanced. A precise definition for the workload balance among N clusters is proposed, as well as a hardware mechanism to estimate it. The proposed RMB steering schemes operate with primary and secondary criteria. Their primary goal is to choose clusters that minimize communication penalties. When there is more than one cluster that fits this criterion, a secondary criterion chooses the least loaded one among them. Our first proposal is the Simple RMB scheme. As a way to reduce communication penalties, it tries to minimize the number of communications required by each instruction, by choosing clusters where most of its source registers are mapped. However, it was found that this scheme produces sometimes large workload imbalances, so it is proposed a second scheme, the Advanced RMB, which includes a balance recovery mechanism: whenever a strong imbalance occurs, the steering ignores the dependence criterion and it strictly chooses the least loaded cluster.

Furthermore, a couple of improvements to these two criteria are proposed. The first improvement is referred to as the Priority RMB (PRMB) scheme [73], and it attempts to reduce communication penalties by partially addressing the different criticality of the source operands: when there are two source registers, and one of them is unavailable, its dependence is prioritized, and the instruction is assigned to its producer's cluster, ignoring other considerations. Even though this scheme may produce more communications than the others, it potentially reduces the critical path length. The second improvement is the Accurate Rebalancing Priority RMB (AR-PRMB). It applies to the cases when there is a strong workload imbalance, and seeks to generate less communications by taking more accurate rebalancing actions: instead of choosing the least loaded cluster - thus totally ignoring dependences - it simply discards the overloaded clusters, which are usually few, before applying the normal criteria with the non-discarded clusters. This algorithm is especially useful for architectures with many clusters because, in case of a strong imbalance, the choice of the least loaded cluster may result too restrictive and generate too many communications. We will show that the AR-PRMB scheme outperforms all other schemes.

1.3.2 Reducing Wire Delay Penalties through Value Prediction

The second contribution of this thesis proposes value prediction as a way to mitigate the penalty of long distance (global) wire delays in general and, in particular, of inter-cluster communications [73]. In general, if a signal must propagate between two distant points within a chip, and if it is possible to generate a prediction of the value to be transmitted, then the receiver may proceed speculatively and the sender will later check the prediction. The actual value will only be transmitted in case of misprediction, otherwise it is sent a simple validity signal, out of the critical-path.

This applies to clustered microarchitectures because copy instructions satisfy all these conditions, so it is proposed to predict the values of remote source registers. Not only unavailable registers are predicted but also those which are available in a different cluster. It is also proposed a new Value Prediction Based steering scheme (VPB) that is aware of the existing prediction mechanism and takes advantage of it to improve workload balance. If a source register is being predicted, hopefully it will require no communication wherever the instruction is sent to. Therefore by ignoring the dependence through this register, the steering scheme may often select the least loaded cluster from a wider choice of clusters. The evaluation shows that value prediction produces significant speedups on a clustered architecture, much larger than for a centralized one, and it also shows that its benefit grows with the number of clusters and the communication latency.

1.3.3 Efficient Interconnects for Clustered Microarchitectures

In a clustered microarchitecture the interconnect is critical for performance because it has important implications on the communication latency. Most previous studies on clustered superscalar architectures assumed either idealized crossbar-like models, or long-latency ring interconnects [3, 52], and focused more on the scalability of the cluster assignment schemes than on the interconnect itself. At a first glance, the most effective design is a crossbar that connects every functional unit output to any other cluster, but this is also the most complex approach, and may have a large impact on the latency of critical operations like issue, register read or result bypass.

Therefore, the third contribution of this thesis proposes several cost-effective point to point interconnects, both synchronous and partially asynchronous, that approach the IPC of an ideal model while keeping the complexity low [75]. The study covers architectures with four and eight clusters, and two technology scenarios where each cluster has two or three adjacent clusters (those at a one hop distance). Issues such as router structures, control flow, contention delays, arbitration of the network links or register file write ports, etc. have a performance impact and are also discussed in detail within our proposals.

Along with the interconnects, we also study adequate cluster assignment schemes. We re-evaluate the above mentioned PRMB and AR-PRMB schemes with the proposed point to point interconnects and show that the AR-PRMB increases its performance advantage as the number of clusters increases because it avoids spreading blindly the instructions among many clusters when the assignment scheme is doing re-balancing actions. We also propose an improved

steering algorithm that is aware of the nonuniform distances of the proposed interconnects to reduce communication latency: instead of minimizing the number of communications, it attempts to minimize the largest required communication distance. We show that this scheme reduces the average communication latency, at the cost of some increase in communications.

Overall, this study shows that point to point interconnects are more efficient than buses, and that the connectivity of the network, together with appropriate steering schemes are key for high performance. It is also shown how these interconnects can be built with simple hardware and achieve a performance close to that of idealized contention-free models.

1.3.4 A Clustered Front-End for Superscalar Processors

As a fourth main contribution, this thesis proposes novel techniques for clustering the main components of the processor front-end [74], i.e. those involved in branch prediction, instruction fetch, decoding, cluster assignment and renaming, with the objective of minimizing replication and inter-cluster communication. The partitioning of some architectural components such as the fetch address generation and the cluster assignment logic need non-obvious solutions because they are part of tight hardware loops, with inputs of one operation being frequently dependent on outputs generated by the previous operation. After partitioning one of these components, if its loop dependences stay global, and cannot be broken into local loop dependences, the global wire delays associated with dependence propagation add to the component's critical path, thus frustrating the goal of reducing its complexity and latency. Effective techniques to minimize the penalty of these communications while avoiding replication are presented, both for partitioning the fetch address generation hardware (including the branch predictor) and the cluster assignment logic.

Clustering the front-end reduces the latency of many hardware structures, which will result in shorter pipelines and/or faster clock rates and translate into significant performance improvements.

1.3.5 Thesis Contributions

The main contributions of this thesis are:

1) On cluster assignment schemes

- It is proposed a family of new algorithms that dynamically identify groups of data-dependent instructions called slices, and make cluster assignment on a per-slice basis. The proposed schemes differ from previous approaches either because they are dynamic and/or because they include new mechanisms to deal explicitly with workload balance information gathered at runtime.
- It is proposed a family of new dynamic schemes that assign instructions to clusters in a per-instruction basis, based on prior assignment of the source register producers, on the cluster location of the source physical registers, and on the workload of clusters.
- It is proposed a definition for the workload balance metric, and mechanisms to estimate it at runtime.

2) On techniques for reducing the penalty of wire delays

- It is proposed to reduce penalties of wire delays through value prediction. This strategy is applied to inter-cluster communications, in a clustered superscalar architecture. It is proven that, because of the large penalties of inter-cluster communications, the benefit of breaking dependences with value prediction grows with the number of clusters and the communication latency.
- It is proposed an enhancement of the cluster assignment scheme that exploits the less dense data dependence graph that results from predicting values to achieve a better workload balance.

3) On the design of a cluster interconnect

- Several cost-effective point-to-point interconnects are proposed, both synchronous and partially asynchronous, that approach the IPC of an ideal model with unlimited bandwidth while keeping the complexity low. The proposed interconnects have much lower impact than other approaches on the complexity of critical components like the bypasses, the issue queues, the register files and the interconnect itself. Included with these interconnect models are some proposals of possible router implementations that illustrate their feasibility with very simple and low-latency hardware solutions.
- For more than 4 clusters, it is proposed a cluster assignment scheme that avoids spreading blindly the instructions among clusters thus producing less communications. It scales better than other proposals with growing number of clusters and communication latency. In addition, a new topology-aware improvement to the cluster assignment scheme is proposed to reduce the distance (and latency) of inter-cluster communications.

4) On the design of a clustered front-end

- Several techniques are proposed to partition the main components of the processor front-end, with the goal of reducing their complexity, and avoiding replication. The proposed techniques minimize the wire delay penalties caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation logic, and the cluster assignment logic.

1.4 Document Organization

The rest of this document is organized as follows:

Chapter 2 describes the clustered superscalar architecture assumed for most of our proposals, its default main parameters, and the experimental framework utilized throughout this thesis, including the simulation methodology and the benchmarks.

Chapter 3 proposes and analyzes several effective dynamic cluster assignment schemes for a cost-effective two-cluster architecture, which are based on the concept of slices. These schemes make use of explicit imbalance mechanisms, so it is also proposed a convenient workload imbalance metric and a mechanism to estimate it at runtime. These proposals are evaluated and compared against previous static and dynamic cluster assignment schemes.

Chapter 4 proposes and analyzes several effective dynamic cluster assignment schemes for a clustered architecture with two or more clusters, which assign instructions to clusters in a per-instruction basis. It is also proposed a new workload imbalance metric that extends the one proposed in the previous chapter to any number of clusters and a new mechanism to evaluate it at runtime. These algorithms are evaluated for architectures with two and four clusters, and compared against previous proposals.

Chapter 5 proposes using value prediction to avoid the penalty of long wire delays, and applies this concept to a clustered microarchitecture in order to reduce inter-cluster communications. It also proposes modifications to the cluster assignment algorithm to improve workload balance by exploiting the dependence graph with less number of edges that results from eliminating dependences.

Chapter 6 investigates the design of on-chip interconnection networks for clustered microarchitectures. It studies some bus and point-to-point interconnects for four and eight clusters, and an improved cluster assignment scheme that seeks to reduce the communication distances. Issues such as router structures, control flow, contention delays, arbitration of the network links and register file write ports, etc. are also discussed.

Chapter 7 proposes effective techniques for clustering the major components of the front-end, such as branch prediction, instruction fetch, decoding, cluster assignment and renaming.

Chapter 8 outlines some of the future steps and open research areas and finally summarizes the main conclusions of this thesis.