DESIGN OF CLUSTERED SUPERSCALAR MICROARCHITECTURES

Joan-Manuel Parcerisa

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (Spain), April 2004

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

DESIGN OF CLUSTERED SUPERSCALAR MICROARCHITECTURES

Joan-Manuel Parcerisa

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (Spain), April 2004

Advisor:
Antonio González

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

ABSTRACT

Over the past decade superscalar microprocessors have achieved enormous improvements in computing power by exploiting higher levels of parallelism in many different ways. High-performance superscalar processors have experienced remarkable increases in processor width, pipeline depth and speculative execution. All of these trends have come at an extremely high increase in hardware complexity and chip resource consumption. Until recently, their main limitation has been the availability of such resources in the chip, but with current technology shrinks and increases in transistor budgets, other limiting factors have become preeminent, such as power consumption, temperature and wire delays. These new problems greatly compromise the scalability of conventional superscalar designs.

Many previous works have demonstrated the effectiveness of partitioning the layout of several critical hardware components as a means to keep most of the parallelism while improving the scalability. Some of these components are the register file, the issue queue and the bypass network. Their partitioning is the basis for the so called clustered architectures. A clustered processor core, made up of several low complex blocks or clusters, can efficiently execute chains of dependent instructions without paying the overheads of a long issue, register read or bypass latencies. Of course, when two dependent instructions execute in different clusters, an inter-cluster communication penalty is incurred. Moreover, distributed structures usually imply lower dynamic power requirements, and they simplify power management via techniques such a selective clock/power gating and voltage scaling.

The purpose of this thesis is to study several key aspects of a clustered architecture with fully distributed components. The first target of this research is the distribution of instructions among clusters, since it plays a major role on performance. The main goals of a cluster assignment algorithm are to keep the workload balanced and to reduce critical communications among clusters. Several techniques are proposed to achieve these goals from two different perspectives: slice-based algorithms, that assign groups of dynamic instructions, and algorithms that operate on a per-instruction basis. The second contribution of this thesis proposes value prediction as a means to mitigate the penalties of inter-cluster communications while also improving workload balance. The third aspect considered is the cluster interconnect, which mainly determines communication latency. Several techniques are proposed to design it seeking for the best trade-off between cost and performance. Finally, the last contribution proposes techniques for clustering the main components of the processor front-end, i.e. those involved in branch prediction, instruction fetch, decoding, cluster assignment and renaming.

AGRAÏMENTS

En primer lloc, gràcies a Antonio González, de qui he après tantes coses sobre l'ofici de la recerca, i sobre l'arquitectura de computadors al llarg d'aquests anys. Gràcies per la seva confiança, els seus savis consells i la seva tenaç iniciativa.

A en Ramon Canal, en Julio Sahuquillo, en José Duato i en Jim Smith, entusiastes col.laboradors amb qui he tingut fèrtils discussions tècniques i que han proporcionat significatives contribucions a aquesta tesi.

A Jordi Cortadella i Mateo Valero per confiar en mi per al primer projecte amb què vaig debutar al DAC, i que va suposar el meu primer contacte amb la recerca científica.

A tots els companys del departament, de qui he rebut sempre mostres de suport i amb els quals he tingut sempre converses estimulants. La llista és llarga, i segur que em deixo algú, però vull esmentar especialment a Pedro, Ramon, Josep-Llorenç i Fernando, amb qui he compartit les millors gresques i hem passat les millors estones. També a en Pepe, Suso, Josep-Ramon, Daniel, Enric, Cristina, Miguel, Agustín, Fermín, Josep, Luisma, Jordi, David, Llorenç, Chema i els antics companys de despatx Susana i Fernando. Gràcies també al personal tècnic de sistemes i de secretaria, que m'han ajudat sempre amb promptitud i eficàcia, i als organismes públics que han finançat aquest treball[1].

A la Roser, per la seva paciència, suport i afecte, amb qui he compartit els moments dolços i els difícils d'aquesta tesi durant tots aquests anys, i amb qui espero compartir nous projectes durant molts anys més.

I finalment, vull agrair a tota la meva família, pares Josep i Manuela i germans Josep, Jordi i Ramon, que sempre m'han guiat i encoratjat, i junt amb els quals he après a conèixer el valor de l'esforç, el plaer de l'aprenentatge, i tantes altres virtuts sense les quals no hauria reeixit. Gràcies en especial al meu pare, que em va brindar l'oportunitat d'iniciar, ja tardanament, els estudis d'informàtica.

1. This work was supported by the Spanish Ministry of Education under contracts TIC-95-0429, TIC-98-0511 and TIC-2001-0995, and by the CEPBA (European Center for Parallelism of Barcelona)

TABLE OF CONTENTS

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.

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

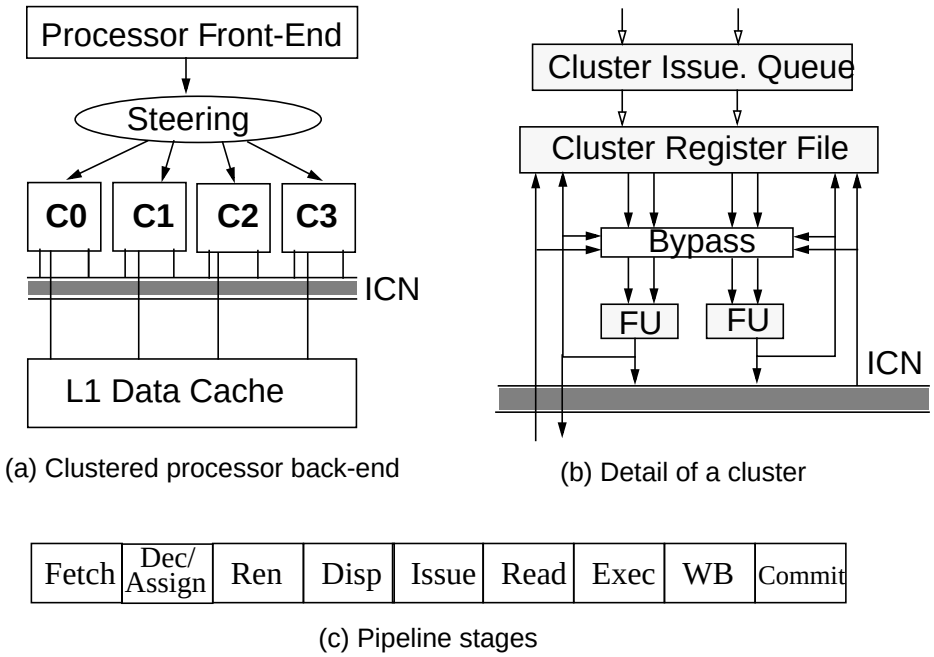


Figure 2-1: Reference Clustered Architecture

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

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.

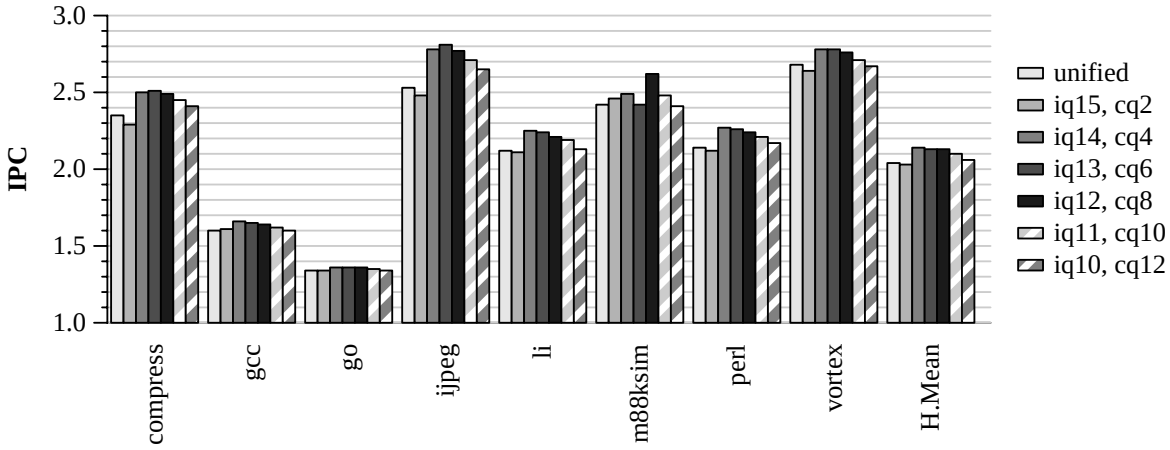


Figure 2-3: Performance improvements of splitting issue queues for copy/regular instructions, for various lengths of the issue queue and copy queue

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.

CHAPTER 3

SLICE-BASED DYNAMIC CLUSTER ASSIGNMENT MECHANISMS

Our first contribution to the design of a clustered architecture is a study of dependence-based partitioning mechanisms that dynamically assign instructions to clusters. Such techniques try to minimize the performance degradation caused by 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 having local resources, and it may prevent a ready instruction from being issued due to the lack of available functional units in its cluster, even though other idle functional units may exist in other clusters. Both kind of delays negatively impact performance if they affect to instructions in the critical path of execution.

In this chapter, we propose and analyze several dynamic cluster assignment schemes which are based on the concept of slices. Slice-based schemes assign the set of instructions involved in a load or store address calculation (Ld/St slice), or in a branch condition calculation (Br slice), to the same cluster, because these instructions are likely to belong to the critical path. For these experiments, we focused on a cost-effective two-cluster architecture (proposed by Palacharla and Smith [68]), although the proposed schemes can also be used in a generic clustered architecture with symmetric clusters. We show that the proposed dynamic slice-based schemes are more efficient than the static one proposed by Sastry, Palacharla and Smith [87], because they adapt better to the run-time conditions and because they address more effectively the workload balance problem. We also show that, with the proposed General Balance scheme, the cost-effective clustered architecture achieves average speed-ups of 35% over a superscalar with similar complexity, while it only achieved average speed-ups of 4% with the static scheme [87].

3.1 Main goals of a Cluster Assignment Mechanism

A clustered architecture exploits the higher simplicity of its components to reduce energy consumption and to permit a faster clock, which are important factors to build high performance computers. The design of the cluster assignment mechanism that distributes the dynamic instruction stream is key for performance in these architectures. In this section we define the two main goals of a cluster assignment mechanism, and we discuss how they are addressed in our proposals, from a conceptual standpoint.

First, due to the partitioning, communications among instructions cannot always use the fast local bypasses, but sometimes they communicate through the slower global paths which may delay the execution of dependent instructions. The use of global bypass paths only in a few cases is an advantage of the clustered approach, since in a centralized architecture all the bypasses are global. Therefore, a primary goal for a cluster assignment algorithm is to minimize penalties caused by inter-cluster communications. Communication delays may degrade performance if the delayed instructions belong to the critical path. In other words, the partitioning goal is to try to execute in the same cluster dependent instructions that belong to a critical path.

Second, due to the partitioning, a clustered processor is less flexible than a centralized one to achieve a uniform utilization of the execution resources. This feature may cause a loss of performance since the amount of execution resources is limited. Thus, it may happen that an instruction in the critical path is delayed by a lack of available resources in its cluster, even though there exists idle resources in other clusters. This additional delay would have been avoided if the steering logic had sent some instructions to a different cluster. We refer to this situation as a workload imbalance among clusters, and since it may potentially degrade the performance, a major goal of the steering logic is to prevent it from happening.

Intuitively, the two main goals, minimal inter-cluster communication penalty and maximal workload balance, are contradictory by nature. Let us just observe that the simplest method for completely eliminating the communications among clusters consists on assigning all the instructions to the same cluster, which is the worst solution for the workload balance goal. Therefore, the task of partitioning involves a trade-off between the two goals that maximizes processor performance. We outline below how these two issues are addressed by the steering logic from a conceptual standpoint. Particular steering techniques are defined in section 3.3.

3.1.1 Communication

The task of minimizing penalties by keeping critical communications local to the clusters requires that the instructions in the critical path are identified, which is a very complex task in the context of a dynamically scheduled processor [31, 107]. Therefore, all the known approaches use some heuristics that target a simpler goal.

The slice-based algorithms are based on the observation that load/store instructions and/or conditional branches are likely to belong to the critical path, because these instructions may

suffer long delays caused by cache misses and branch mispredictions, respectively. Thus, a goal of the proposed steering scheme is to assign the chain of instructions involved in every load address and/or branch condition calculations to the same cluster.

3.1.2 Workload Balance

Obtaining an optimal partitioning that minimizes the delays of critical instructions caused by workload imbalance is a hard problem, due to the difficulty of determining which instructions belong to the critical path, and which will be the availability of functional units when they become ready. The problem is even more complex due to the lack of a well established metric for the intuitive concept of workload balance. All the existing algorithms, as well as those proposed here resort to heuristics to address this problem. In any case, workload balancing should be performed with minimal impact on the communication overhead.

A first naive approach is a random assignment to either cluster where all clusters have the same probability of being selected. A second approach detects when there is a workload imbalance and how much unbalanced it is, and also determines which is the least loaded cluster. There are many alternatives to determine at run-time the individual workloads of the clusters and their relative imbalance, because there is not a unique definition.

The workload imbalance may be estimated by counting the difference in the number of instructions steered to each of the two clusters (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 instructions 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 chapter considers the two metrics (I1 and I2) by maintaining a single integer imbalance counter that combines the two informations. Each cycle, this counter is updated by adding the I1 metric to the average of I2 with the previous values of the counter along the last N cycles (we have determined empirically that 16 is an adequate value for N).

Although the schemes presented here use the above combination of I1 and I2, 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, since metric I1 alone gives performance figures quite close to those produced by the combination of I1 and I2, it could be used alone in a particular cost-effective implementation. On the other hand, since I2 matches more closely the concept of imbalance as described in the beginning of section 3.1, I2 will be used to report the workload imbalance of the experiments.

3.2 The Cost-Effective Clustered Microarchitecture

The slice-based code partitioning schemes presented and evaluated in this chapter are built upon the early proposal of Palacharla and Smith [68], also developed in collaboration with Sastry [87]. In those papers it is described how a typical superscalar with an integer and a FP subsystem may benefit from augmenting the latter with a few ALUs capable of executing simple integer operations. Since exploiting the new features requires that some integer instructions are steered to the FP subsystem, they propose a compile-time cluster assignment method based on the concept of "program slices" [87]. In order to facilitate comparisons, our proposals are evaluated with a similar architecture, although they could be applied to any other clustered architecture. This architecture will be referred to as the cost-effective clustered architecture, and is briefly described below.

The motivation for the cost-effective clustered microarchitecture starts from the observation that many current superscalar processors are already partitioned into two subsystems or clusters, the integer and the floating point one, but the whole FP subsystem remains idle during the execution of integer programs. Therefore, the FP subsystem can be easily extended to execute simple integer and logical operations (no multiplication and division), which represents a very small added hardware cost considering today's transistor budgets. The advantage of the new architecture is that its floating-point registers, data path and mainly, its issue logic are used for any type of application.

The cost-effective clustered architecture we used to conduct our experiments is very similar to the one mentioned above (see figure 3-1). It consists of two clusters, each containing a register file, an issue queue, and some basic integer and logic functional units. Moreover, one of them, which will be referred to as the integer cluster, contains also complex integer functional units (multiplier and divider), while the other cluster, referred to as the FP cluster, contains also the floating-point functional units. There is also a communication datapath to copy values from one cluster to the other. Loads and stores are divided into two operations. One of them calculates the effective-address and the other accesses memory. The effective-address is computed in an adder, in either cluster, and then it is forwarded to the access instruction, in the common disambiguation hardware. 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.

Our cost-effective clustered architecture differs from the one proposed by Palacharla et al. [68] in the following aspects. First, since our steering schemes are dynamic, there is a piece of hardware, referred to as the steering logic, to perform the cluster assignments and monitoring the workload of clusters. Second, the register rename table needs to be extended to support up to two mappings per logical register, one for each cluster. Third, load and store address calculations may execute in either cluster, which removes one important constraint of the previous proposal.

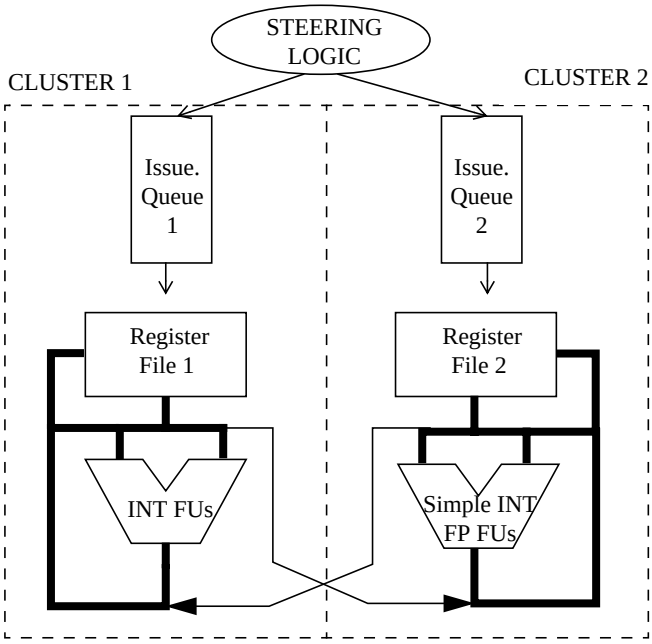


Figure 3-1: Block diagram of a cost-effective clustered architecture

Finally, looking at the above description, one realizes that our cost-effective clustered architecture is almost just a particular instance of our Reference Clustered Architecture with the only difference that the two clusters are heterogeneous. Since all the rest of mechanisms that support the clustered approach - map table extensions, copy instructions, inter-cluster bypasses, etc - are identical to those of the Reference Clustered Architecture, we omit here a full description (see chapter 2 for details).

3.3 Cluster Assignment Schemes

This section presents several new cluster assignment schemes and evaluates their performance. We first define some terminology and describe the experimental framework. Then, we compare the effectiveness of a static assignment versus a simple dynamic mechanism. Finally, other alternative dynamic schemes are presented and evaluated.

3.3.1 Terminology

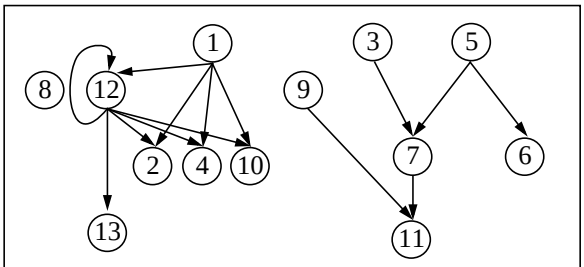
A register dependence graph (RDG) represents all register dependences in a program. It is a directed graph that has a node associated to each static instruction and an edge for every data dependence (true dependence) through a register. Memory instructions are special cases since they are split into two disconnected nodes, one representing the address calculation and the other the memory access. Figure 3-2 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.

```
for (i=0;i<N;i++) {
     if (C[i]!=0)
          A[i]=B[i]/C[i];
     else A[i]=0;
}
```

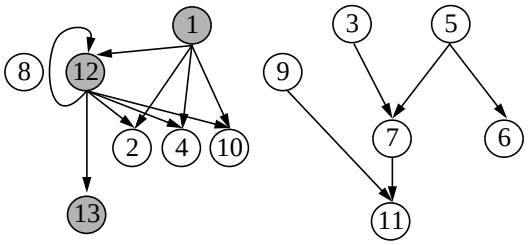
Code in C

```
1            MOV  0, Ri
2    for:    EA   RB+Ri
3            LD   RBi
4            EA   RC+Ri
5            LD   RCi
6            BEQZ RCi, l1
7            DIV  RBi, RCi, RAi
8            JMP  l2
9    l1:     MOV  0, RAi
10   l2:     EA   RA+Ri
11           ST   RAi
12           ADD  Ri, 4, Ri
13           BNEQ Ri, N*4, for
```

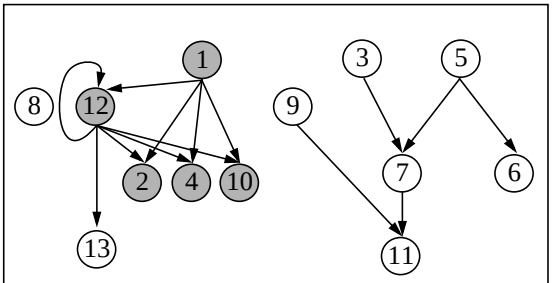
Assembly code (R*A*, R*B* and R*C* contain the initial addresses of arrays A, B, C respectively)



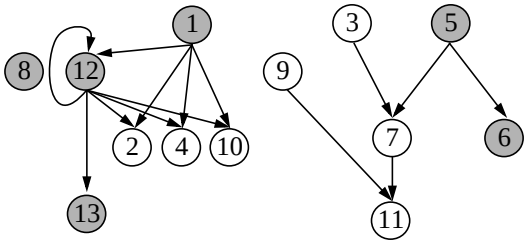
Register dependency graph



Backward slice with respect to node 13 (shaded nodes)



LdSt slice (shaded nodes)



Br slice (shaded nodes)

Figure 3-2: Example of a RDG

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* [87]. Figure 3-11 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 3-2 shows the LdSt slice and the Br slice of the sample program.

3.3.2 Experimental Framework

Performance figures were obtained through a cycle-accurate timing simulator based on the SimpleScalar tool set v3.0 [15], which was extended to simulate the architecture described in section 3.2. Results are presented for the SpecInt95 benchmark suite. Table 3-1 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 3-2 shows the architectural parameters of the assumed processor.

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 3-1: Benchmarks and their inputs

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 3-2 except that it has neither integer units in the FP cluster nor inter-cluster bypasses.

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

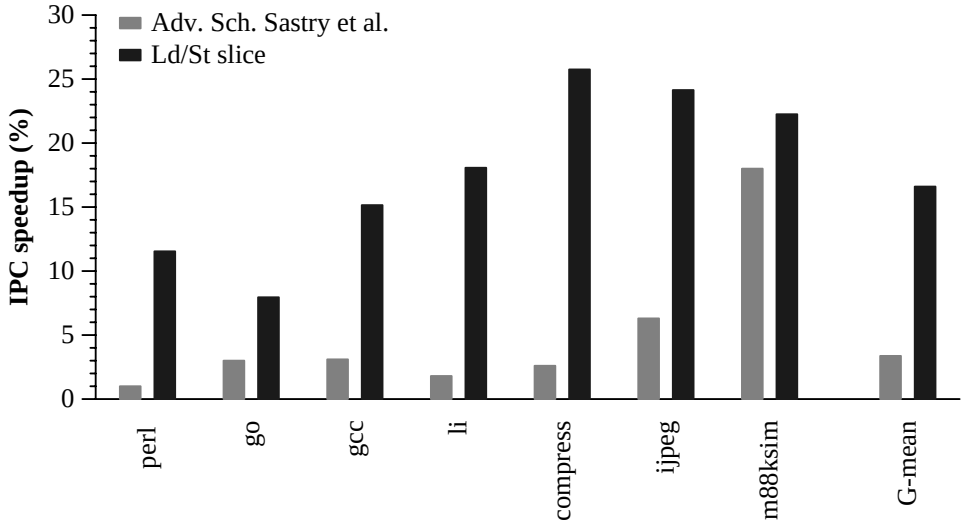


Figure 3-3: Static versus dynamic LdSt slice steering

3.3.3 Static versus Dynamic Cluster Assignment with LdSt Slice Steering

Cluster assignment can be done at compile time (static) or at run time (dynamic). The first method relies on the compiler, which allocates each static instruction to a cluster, while the second method is based on a specialized hardware that decides where to dispatch each dynamic instruction. The main advantage of a static assignment is that it requires minimal hardware support, but its downside is that it requires to recompile the applications because it extends the ISA for encoding the steering information. Furthermore, recompilation is needed for each new microarchitecture generation that changes the relevant features of the clusters.

In contrast, a dynamic assignment method does not require to recompile, because it makes clustering transparent to the compiler. In addition, the information used by the dynamic steering logic (workload balance, data dependences) is obtained directly from the actual pipeline state, rather than estimations of the compiler. Therefore, a dynamic steering scheme is more effective than a static approach because it is more adaptable to the actual processor state. This work focuses on this type of steering.

The static assignment proposed by Sastry *et al.* [87] is based on sending all instructions that belong to the subgraph defined by the LdSt slice, probably extended with neighbor instructions, to the integer cluster. This extension is based on some heuristics that try to approximate its effect in terms of workload balance and communication overheads.

We have evaluated the speed-ups of the cost-effective architecture over a conventional architecture (one without the simple integer units added to the FP cluster). Figure 3-3 compares the speed-ups of Sastry's et al. static assignment with the speed-ups achieved by a simple dynamic assignment 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 assignment scheme as *LdSt slice steering*. This dynamic assignment

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.

For the experiments in figure 3-3, the numbers for the static assignment have been obtained from the original paper [87] 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 assignment speed-up is just 3%.

3.3.4 LdSt Slice Steering versus Br Slice Steering

The performance of any assignment 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 assignment scheme 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 suggests an alternative assignment 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 3.3.3 for the LdSt slice steering.

Figure 3-5 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 3-4. 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. Figure 3-6 shows the distribution function of the I2 workload imbalance metric, i.e., the difference between the number of ready instructions in each cluster for each cycle (see section 3.1.2). It can be seen that both dynamic assignment 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

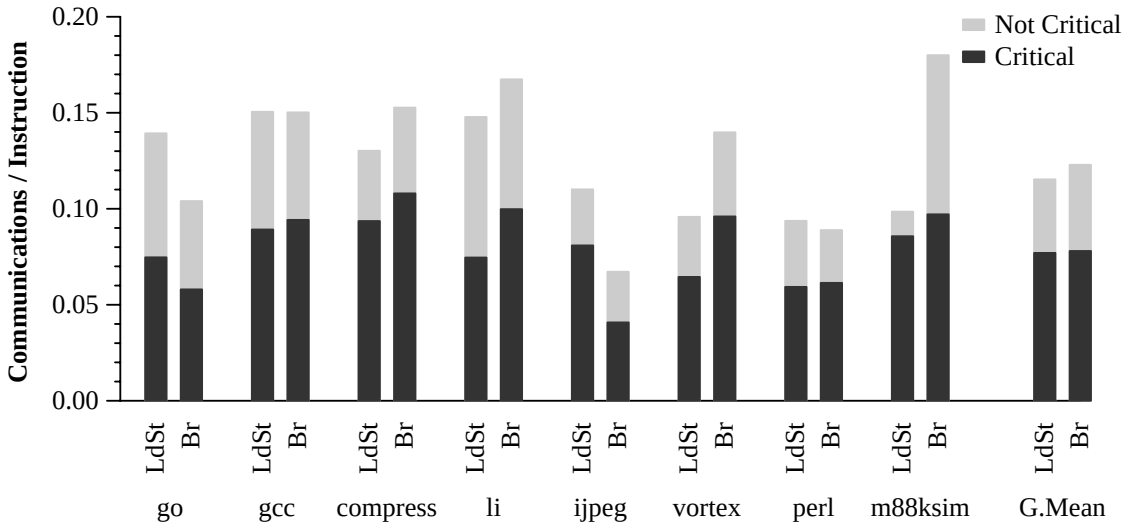


Figure 3-4: LdSt slice versus Br slice steering: communications per instruction

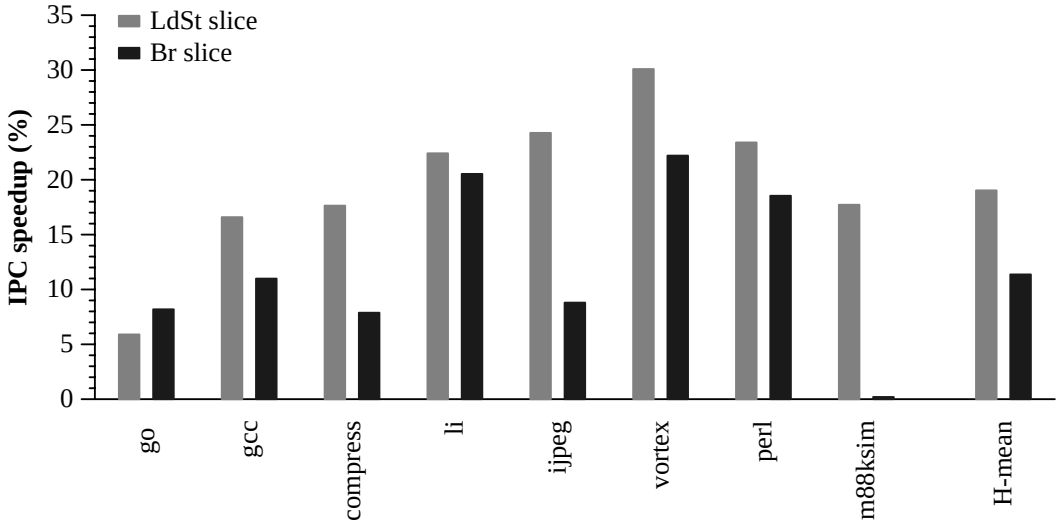


Figure 3-5: LdSt slice versus Br slice steering: performance

(resp. Br slice) were dispatched to the integer cluster. This motivates the next assignment scheme.

3.3.5 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 the absolute value of the workload imbalance counter (see section 3.1.2) exceeds a given

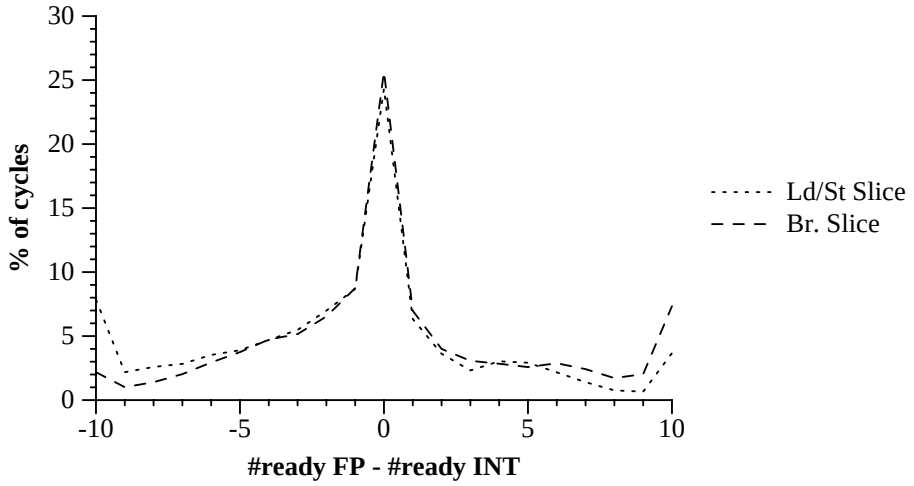


Figure 3-6: LdSt slice versus Br slice steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)

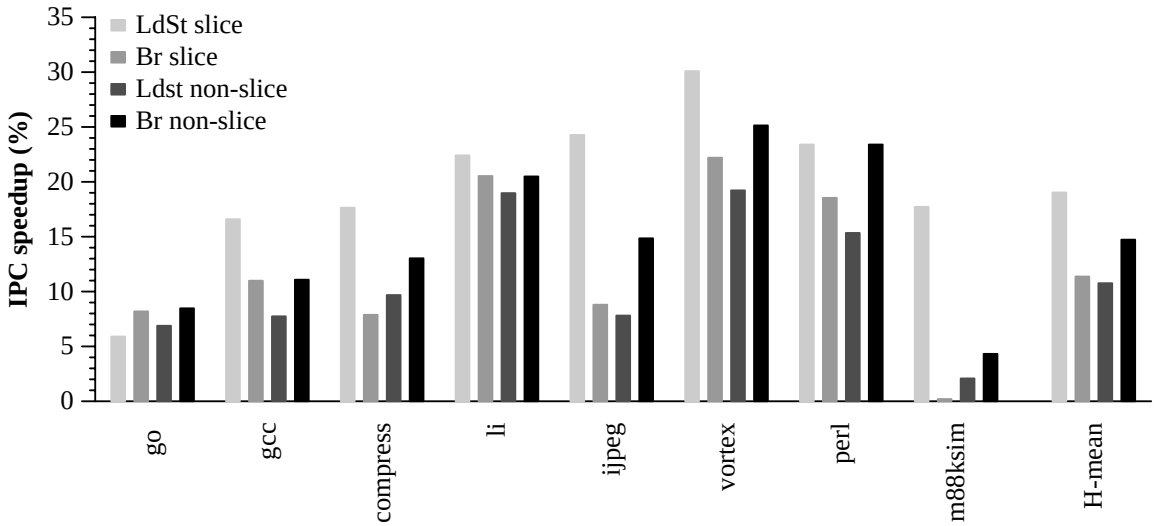


Figure 3-7: Non-slice Balance steering versus slice steering: performance

threshold. 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. We have empirically determined that a threshold of 8 is adequate for the cost-effective architecture.

Figure 3-7 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 3-8. This figure shows that the non-slice balance steering significantly increases the number of communications for the LdSt slice whereas it has about the same number of communications as the slice steering for the Br slice.

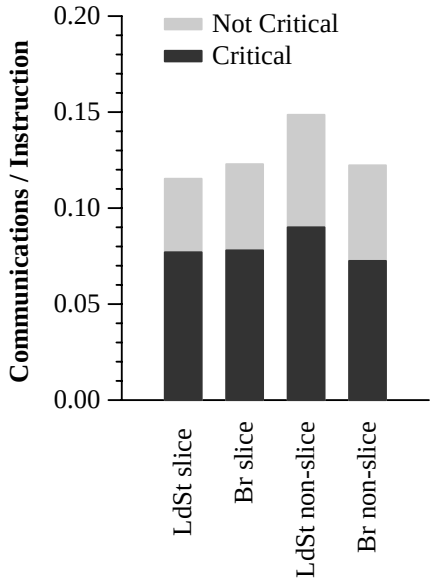


Figure 3-8: Non-slice Balance steering versus Slice steering: number of communications per dynamic instruction (SpecInt95 average)

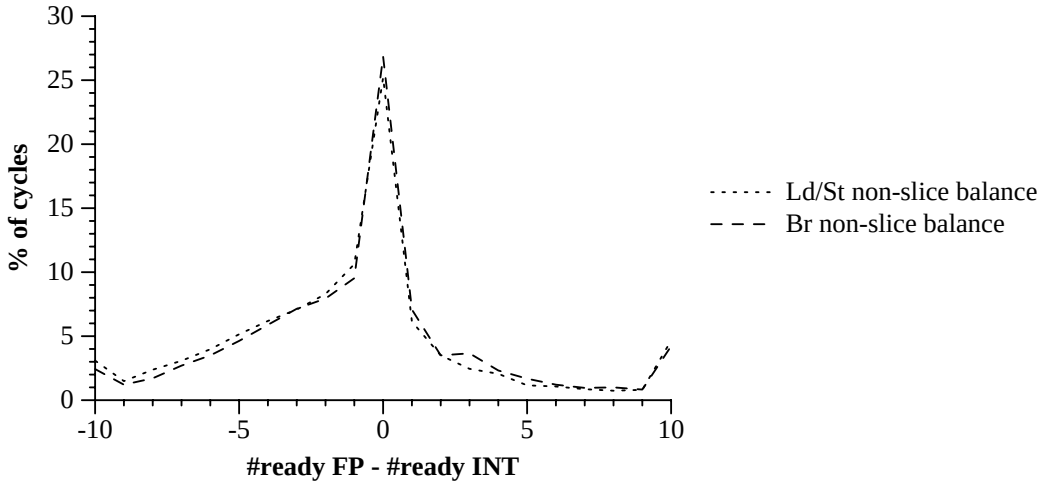


Figure 3-9: Non-slice Balance steering versus slice steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)

Figure 3-9 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 3-6), 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 assignment scheme.

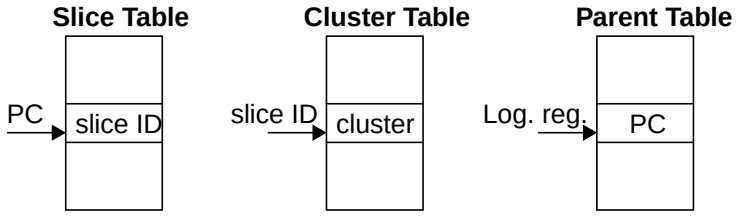


Figure 3-10: Hardware support for the Slice Balance steering

3.3.6 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 backward 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 3-10. 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 3-11 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 3-12 shows the distribution of the I2 workload imbalance metric for the slice balance steering (LdSt and Br) and compares it with that of a naive 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

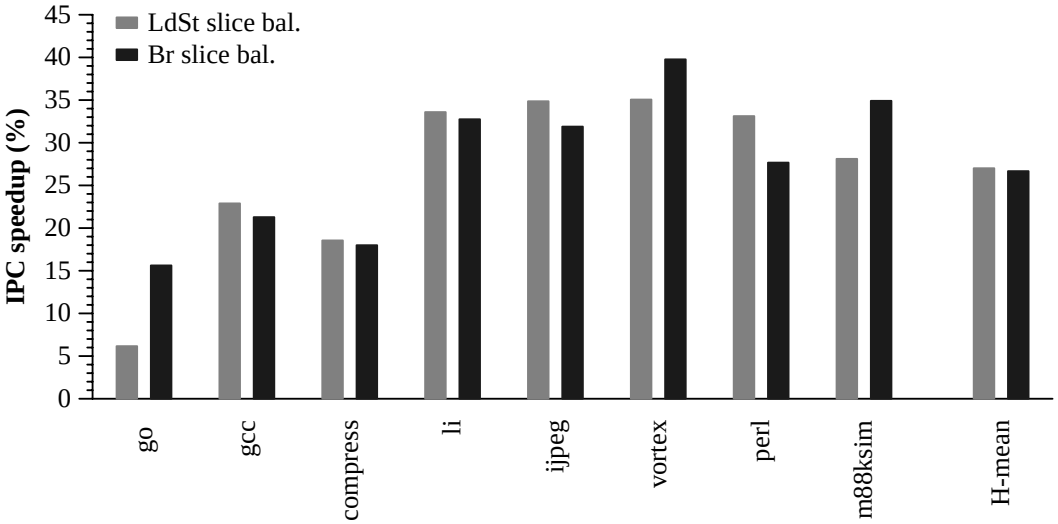


Figure 3-11: Slice Balance steering: performance

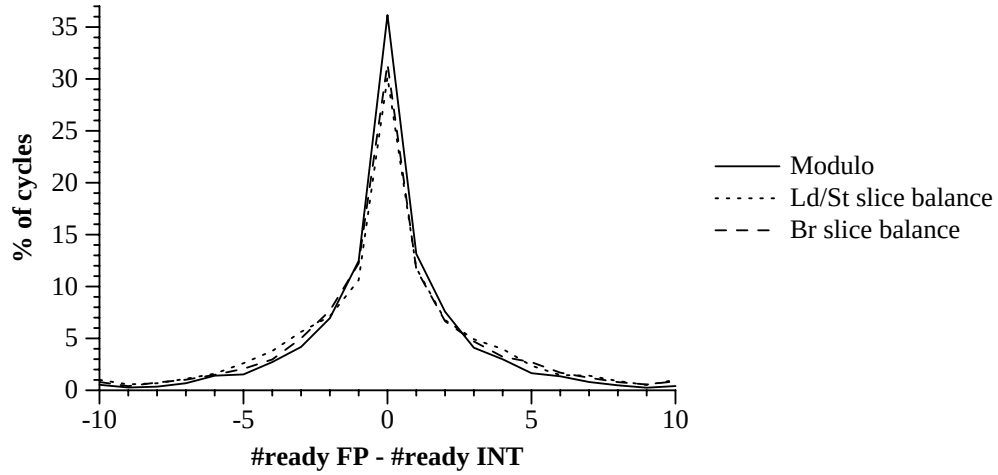


Figure 3-12: Slice Balance steering: distribution of the workload imbalance, metric I2 (SpecInt95 average)

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.

3.3.7 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

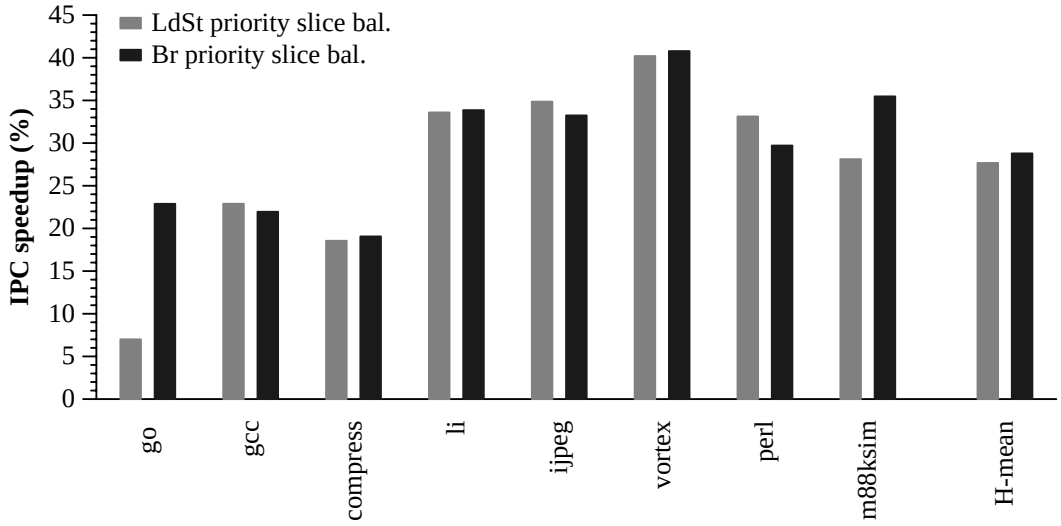


Figure 3-13: Priority Slice Balance steering: performance

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 such. 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 3.1.2 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 3-10) 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.

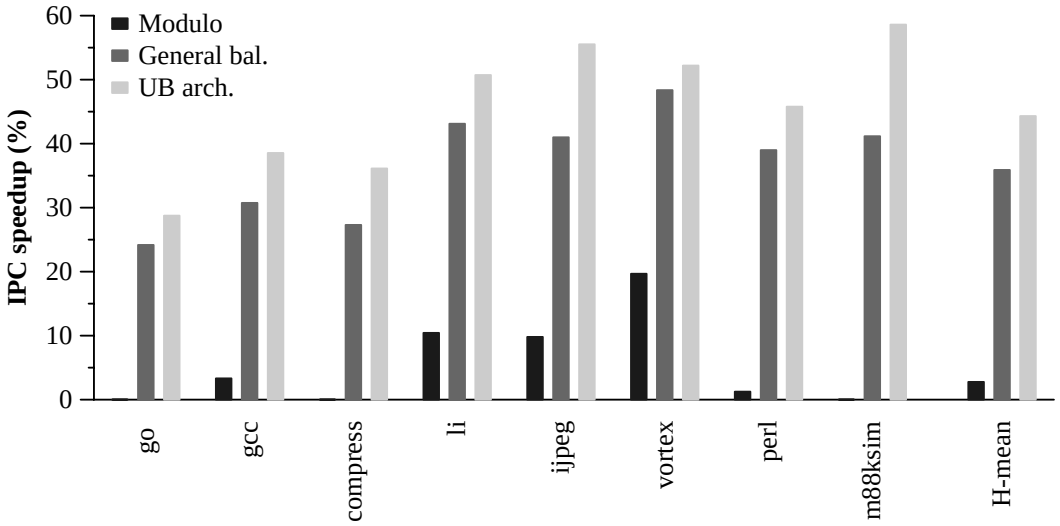


Figure 3-14: General Balance steering

Figure 3-13 shows the performance of the priority slice balance steering. It achieves an average speed-up 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 3-11). 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.

3.3.8 General Balance Steering

The *general balance* steering is a particular case of the previous steering scheme, in which the criticality threshold is set so high that there are no critical instructions, 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 program slices (see figure 3-10) is not needed and no extra hardware is required to detect the criticality of instructions. Actually, the general balance scheme falls into a different category, the instruction-based schemes, rather than slice-based, because it makes all steering decisions at a single instruction granularity. Thus, the performance of this scheme is analyzed here to provide comparison with the rest of the slice-based schemes, but it will be analyzed in more detail in chapter 4 (where it is called the Advanced RMB scheme), in the context of our Reference Cluster Architecture.

Figure 3-14 shows the performance of the *general balance* scheme. It also includes the performance of the *modulo* steering (see section 3.3.6) and that of a conventional 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 assignment 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

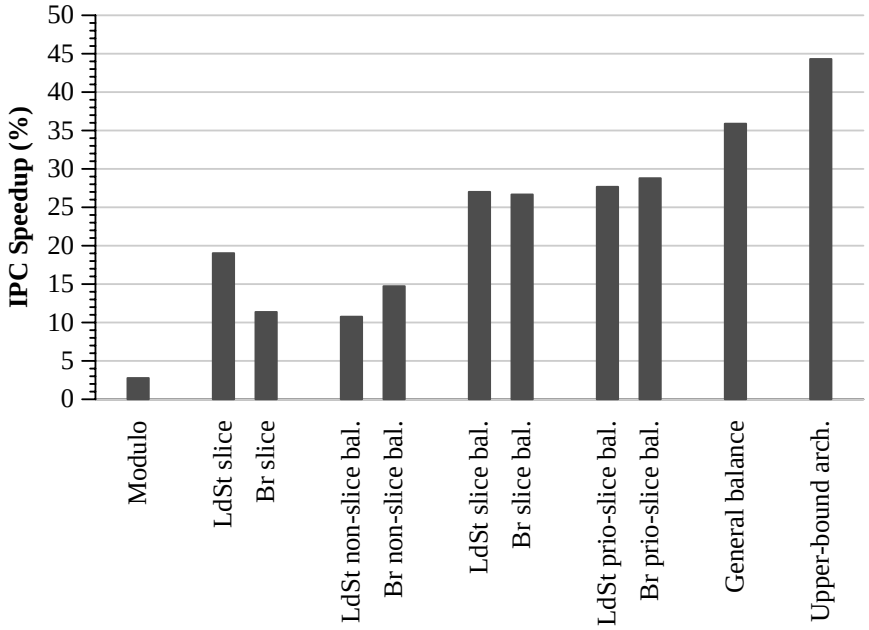


Figure 3-15: Performance comparison among all the proposed steering schemes

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).

3.4 Evaluation

In this section it is summarized the performance results of all the above proposed cluster assignment schemes. Next, the best performing scheme is compared to another dynamic steering scheme previously found on the literature, and it is analyzed the suitability of the cost-effective architecture for FP programs.

3.4.1 Overall Performance Comparison

Figure 3-15 depicts, for each steering scheme, the SpecInt95 average speedups of the cost-effective architecture over the base architecture. The graph also includes, for comparison, the performance of the *upper bound* architecture (see section 3.3.8). The best slice-based scheme is the priority slice balance, which achieves a speedup of 28%, but the best performance is obtained with the general balance steering scheme, which achieves an average speedup of 36%, just 8% smaller than the upper bound. In other words, this steering scheme enables the cost-effective architecture to realize most of its potential, with integer programs.

These results show that in a cost-effective architecture, with an appropriate dynamic steering like our *general balance* scheme, idle floating-point resources can be profitably exploited to speed-up integer programs, with minimal hardware support and no ISA change.

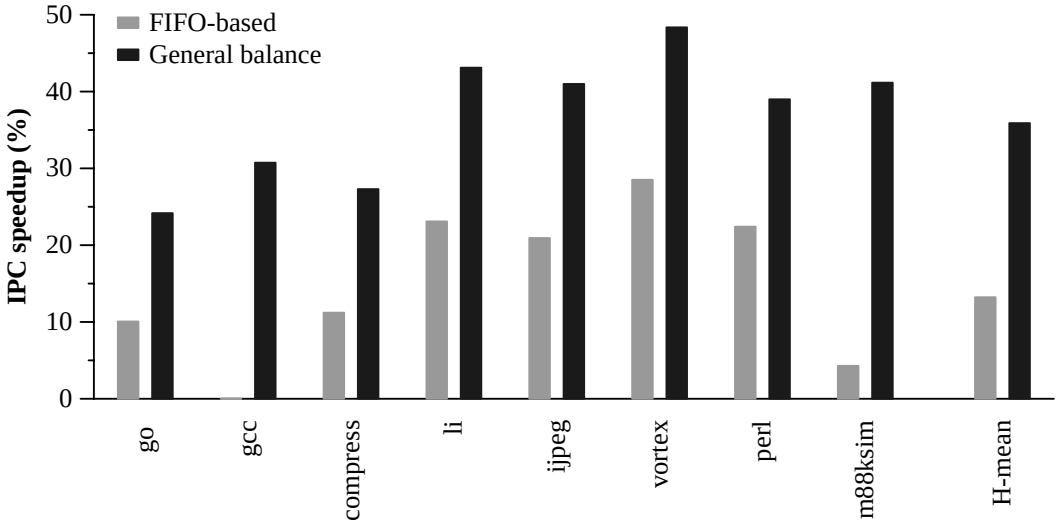


Figure 3-16: General Balance steering versus FIFO-based steering [70]

3.4.2 Comparison to a Static Slice-Based Scheme

Sastry, Palacharla and Smith [87] proposed a static slice-based assignment algorithm, similar to our dynamic LdSt slice scheme. Both schemes were compared in section 3.3.3 (figure 3-3), and the results showed that the dynamic approach is much more effective than the static one: the speed-ups of the dynamic scheme are up to 10 times bigger than those of the static scheme, for several reasons: first, because a dynamic steering technique does a better job not only at reducing inter-cluster communication but also workload imbalance, which was already reported to be one of the main drawbacks of the static approach [87]; second, because a dynamic steering adapts more accurately to many run-time conditions that are difficult to estimate at compile time.

From these results and those of the previous section we conclude that all the proposed schemes significantly outperform the previous static slice-based proposal.

3.4.3 Comparison to Another Dynamic Scheme

Palacharla, Jouppi and Smith [70] proposed a dynamic assignment approach, for a different clustered architecture, that could also be applied to our assumed architecture. Their basic idea is to model each issue queue as if it was a collection of FIFO queues. Instructions are steered to FIFOs following some heuristics that ensures that any two consecutive instructions in a FIFO are always dependent. If all the source registers are ready, or their producers do not stay at any of the FIFO tails, then the instruction is dispatched to an empty FIFO. The assignment policy chooses always empty FIFOs from the same cluster until all of them are used, then it switches to another cluster (for more details refer to the original paper [70]).

For the following experiment, we modelled the FIFO-based steering with 8 FIFOs per cluster, each with 8 entries (thus having a total of 64 scheduler entries per cluster), on our

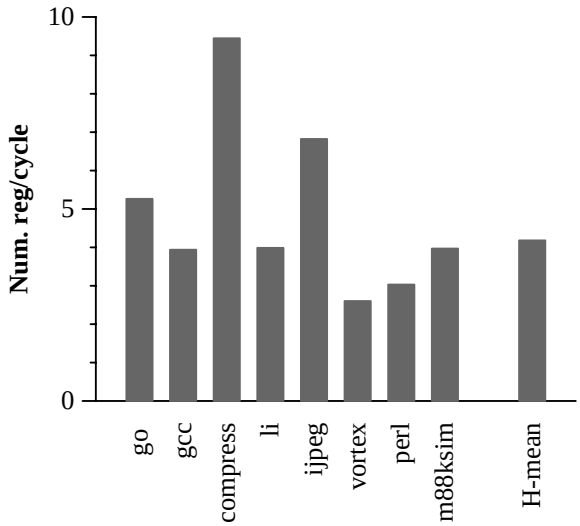


Figure 3-17: Register replication on a cost-effective 2-clusters architecture

assumed cost-effective architecture. Figure 3-16 compares the performance of the general balance scheme versus the FIFO-based scheme. The results show that the general balance steering 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.

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. Note that the FIFO-based scheme does not usually dispatch an instruction to the cluster that produces its operands if they are ready, or their producers are not at some FIFO tails.

3.4.4 Register Replication

As outlined in section 3.2, the cost-effective 2-clusters microarchitecture requires some degree of register replication. We have evaluated the average number of logical registers that have a physical register allocated in both clusters, with the general balance steering scheme. The results in figure 3-17 show that the required register replication is very low. Instead of replicating the whole physical register file, as for instance the Alpha 21264 processor does [53], this architecture requires on average 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 [29, 70].

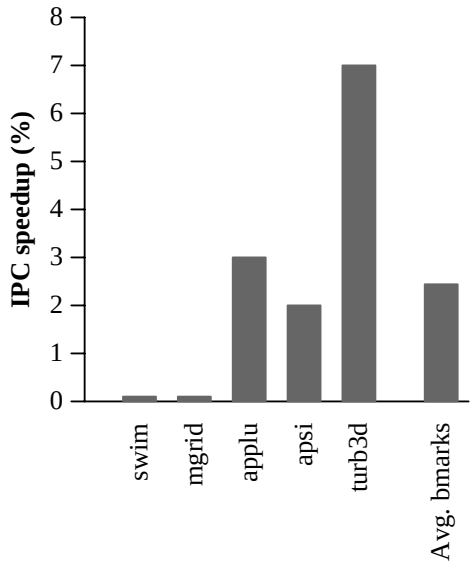


Figure 3-18: Performance of the SpecFP95 on a cost-effective 2-clusters architecture

3.4.5 Running FP Programs on a Cost-Effective Clustered Architecture

So far, all the evaluations of the cost-effective clustered architecture included only integer benchmarks because the particular optimizations of this architecture specifically target this kind of applications. However, we want to investigate whether sending integer instructions to the FP cluster may degrade the performance of floating-point programs due to the sharing of resources in that cluster.

We have measured the performance of the cost-effective clustered architecture with the general balance scheme for several SpecFP95 [97] benchmarks. Figure 3-18 shows the speed-ups of the cost-effective architecture. We can see that none of the programs is slowed down and even in some of them the speed up is significant (7% in turb3d). On average, floating point programs perform a 3.2% better. When the FP cluster has a high workload (its resources are heavily demanded by FP instructions), the balance mechanism will refrain the steering logic from sending integer instructions to that cluster, so that they do not compete for the FP resources. On the other hand, in periods of few FP calculations, the balance mechanism will send some integer instructions to the FP cluster and we could expect some speed-ups in this case.

3.5 Conclusions

We have proposed a number of slice-based mechanisms that dynamically partition a sequential program into the different clusters of a clustered microarchitecture. The slice-based schemes assign clusters to some groups of dependent instructions called slices. A similar concept was used in some early static code assignment scheme that attempted to exploit the potential of a

cost-effective two-clustered architecture with the capability to execute simple integer instructions in both the integer and the FP datapaths [87]. Thus, in order to facilitate comparisons, our evaluations focus on a similar cost-effective microarchitecture, and we show that our dynamic schemes outperform the previous static proposal, due to the much better ability to keep cluster workloads balanced.

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 of clusters. We have shown that all the schemes provide a significant speedup over a conventional 8-way microarchitecture (4int + 4fp). For instance, the general balance steering achieves an average speed-up of 36% for the SpecInt95, and just an 8% below an upper bound. In other words, it enables the cost-effective architecture to realize most of its potential, with integer programs. We have also shown that this scheme significantly outperforms a previous dynamic proposal [70], because it generates a significantly lower number of inter-cluster communications.

Our results also prove that, with a cost-effective architecture, idle floating-point resources can be profitably exploited to speed-up integer programs, with minimal hardware support and no ISA change. Moreover, we show that such a cost-effective architecture also performs better than a conventional architecture when running the SpecFP95 benchmarks, even though the FP-cluster datapath is shared by integer and FP instructions.

Finally, we also show that with our steering schemes, the distributed register file has very few replicated registers. Overall, each cluster's register file has few ports and few registers, which results in a very low latency register file with very low power dissipation.

CHAPTER 4

INSTRUCTION-BASED DYNAMIC CLUSTER ASSIGNMENT MECHANISMS

In this chapter we propose and analyze several dynamic cluster assignment schemes that will be referred to as the *instruction-based steering schemes*. These schemes work at a finer granularity than the slice-based schemes studied in the previous chapter, because they assign instruction per instruction to clusters instead of considering instruction groups. In both cases, the main steering decisions are based on the dependences through registers but, unlike the slice-based schemes, the instruction-based ones only consider the immediate dependences of every dynamic instruction with its predecessors in the data dependence graph.

Our main contribution is the Advanced RMB scheme, which assigns instructions following primary and secondary criteria. The primary criterion selects clusters requiring the fewest communications to read the source registers. If more than one cluster is selected, the secondary criterion chooses the least loaded one. However, the algorithm ignores the primary criterion in case the workload imbalance exceeds a given threshold. We also present two improvements to this algorithm. The first one is the Priority RMB scheme, that slightly modifies the primary criterion: when any source register is unavailable it gives priority to the producer cluster of the unavailable source register. The second one is the Accurate Rebalancing Priority RMB scheme, that slightly modifies the action to be taken when the workload imbalance exceeds the threshold: instead of totally ignoring the primary criterion, it just excludes the overloaded clusters prior to applying the two criteria.

In this chapter, our instruction-based schemes are evaluated in the context of our Reference Cluster Architecture (see Chapter 2), with two and four clusters, and they are compared to the best previously proposed schemes. It is shown that the proposed Accurate Rebalancing Priority RMB scheme significantly outperforms the best previously proposed schemes, since it achieves the best trade-off between communications and workload balance.

4.1 Communication and Workload Balance

In the previous chapter we described the two main goals of a cluster assignment mechanism: minimizing the penalties of inter-cluster communications and maximizing the workload balance. In this section it is described how these two important issues are addressed by the instruction-based partitioning schemes that are proposed.

4.1.1 Communication

Inter-cluster communication penalties may occur if two dependent instructions are steered to different clusters, and they belong to the critical path of execution. Since determining the critical path is a very complex task in the context of a dynamically scheduled processor [107], all the known approaches use some heuristics that approximate this objective. Most of the presented schemes try to simply minimize the number of inter-cluster communications.

In the context of an architecture with distributed register files, like the Reference Cluster Architecture, the instruction results are by default stored only in the register file of the cluster where the instruction executes. Whenever a communication is required to read a source operand, the dispatch logic generates a copy instruction, allocates a new physical register to the same logical register, and updates the map table to show its multiple mappings with the valid bit set. The valid bit associated to each field of the register map table (see chapter 2) indicates whether a source register may be directly read in the corresponding cluster without requiring a communication. The steering schemes proposed in this chapter use this information to minimize communications, hence we refer to them as *register mapping based* (RMB).

Note also that we only analyze steering schemes for architectures with a distributed register file. On some other architectures, the register file is replicated in all clusters, and all copies are kept consistent by broadcasting every result to all clusters (e.g. the Alpha 21264 [53]). In such architectures, the inter-cluster communication latency is hidden if the value has already been produced and broadcast by the time it is needed by the consumer. Thus, the task of minimizing communication penalties is slightly different, since it must take into account only the source operands that are not yet broadcast. While this architectural approach simplifies the steering mechanism, it shifts the complexity into the register file and the bypass network, and will not be analyzed further since it falls beyond the scope of our work.

4.1.2 Workload Balance

In section 3.1 the problem of the workload imbalance among clusters was described as the situation where an instruction is delayed by the lack of available functional units in its cluster, even though there exists idle functional units in other clusters. Since workload imbalance may potentially degrade performance, a major goal of the steering logic is to prevent it from happening. As mentioned in section 3.1.1, obtaining an optimal partitioning that minimizes the delays of critical instructions caused by the workload imbalance is a hard problem, and all the existing algorithms, as well as those proposed here, resort to heuristics to address it.

There are many alternatives to determine at run-time the individual workloads of the clusters and their relative imbalance, because there is not a unique definition. In section 3.1.2 the workload imbalance was reported in terms of metric I2, which was defined as the difference in number of ready instructions between the two clusters. However, such a definition is not suitable for an architecture that may have more than two clusters, like our Reference Cluster Architecture. Therefore, in this chapter it is replaced by a new definition, though similar in concept. From the description of the workload imbalance in the previous paragraph, we intuitively define the workload imbalance at a given instant of time as the total number of ready instructions that cannot issue, but could have issued in other clusters since they have idle functional units. We will refer to this figure as metric NREADY, and it is used to report the workload imbalance in our experiments. To be more precise, if we count for each cluster the difference between the number of ready instructions and its issue width; next we sum separately the positive and the negative differences; then NREADY is defined as the minimum between the absolute values of these two sums.

In section 3.1.2, we found that the steering decisions were best guided by another metric, which was mainly influenced by the number of instructions dispatched to each cluster (metric I1). Unfortunately, as it was defined, I1 is not suitable for an architecture that may have more than two clusters, like our Reference Cluster Architecture. Therefore, in this chapter it is replaced by a new definition, though similar in concept, that will be referred to as metric DCOUNT. We tried also with metrics based on the number of instructions present in the issue queues, but DCOUNT was found to give the best performance. From a conceptual standpoint, DCOUNT may be defined as the maximum of the absolute deviations of the accumulated number of dispatched instructions per cluster. Despite this apparently complex definition, it may be implemented with reasonably low complex hardware: the processor has a signed counter for each of the N clusters that measures its workload. Its value is initially zero, and it is updated in the following way: for every instruction dispatched to a cluster, the corresponding counter in that cluster is increased by N-1, while the other N-1 counters are decreased by 1 (i.e. the sum of the counters is kept always zero). Therefore, the value stored in the counter of a given cluster is N times the difference between the total number of instructions dispatched to that cluster and the average number of instructions dispatched per cluster (the deviation). The workload imbalance is calculated as the maximum absolute value of the workload counters. Note also that in the case of two clusters, DCOUNT is equivalent to I1, and a single counter will suffice. On a branch misprediction, the pipeline is flushed, and the precise state of the workload counters needs to be recovered. However, we found that just clearing all counters performs quite well, and requires much less complex hardware.

The NREADY figure matches more exactly our definition of workload balance. However, when it is used by the steering logic, the actions taken to compensate a workload imbalance (sending instructions to the least loaded cluster) may not update immediately the NREADY figure, if some of the steered instructions are not ready. When this occurs, the corrective action may result disproportionate, and cause an imbalance in another direction or some unnecessary inter-cluster communications. This does not happen with the DCOUNT figure, since it varies instantly and in proportion to the steering decisions, which allows the steering logic to gauge more accurately the actions to compensate a workload imbalance. Thus, the steering logic uses

the DCOUNT figure to determine balancing actions and we use the NREADY figure to measure and report workload balance.

4.2 Instruction-Based Cluster Assignment Schemes

This section presents and evaluates several instruction-based dynamic cluster assignment schemes. This kind of algorithms are finer-grain than slice-based ones, as they work instruction by instruction instead of considering instruction groups. On the one hand, they ignore the relationship with other instructions that cooperate in the calculation of the same load address or branch condition unless they are directly dependent, but the finer granularity make them more flexible to assign instructions to clusters.

All the proposed RMB steering algorithms try to minimize the number of communications required by an instruction by choosing a cluster where the highest number of its source operands are currently mapped. Since this criterion is included in all schemes, we outline its rules here once, to improve readability:

- If the instruction has no register source operands, all the clusters are considered as potential clusters to dispatch the instruction.
- If the instruction has one register source operand, only those clusters where it is mapped are considered.
- If the instruction has two register source operands and there is at least one cluster where both are mapped, those clusters that have both operands mapped are considered; otherwise those clusters where one of the registers is mapped are considered.

Note that this steering criterion may not deliver always a unique cluster selection, so another secondary criterion must be considered to further refine it.

4.2.1 Experimental Framework

The experiments in this chapter evaluate the instruction-based schemes assuming the Reference Clustered Architecture and all the experimental setup described in chapter 2, including the simulator, benchmarks and architectural parameters. In this section, for simplicity, we assume a configuration with four clusters and the SpecInt95 benchmark suite [97], but in section 4.3.5 we study configurations with two and four clusters and they are evaluated using both the SpecInt95 and the Mediabench [56, 62] benchmark suites.

The IPC of the clustered architecture is compared to the IPC of a centralized architecture because it is an upper bound reference that helps to see how much IPC they are trading-off for clock speed and power savings. The following subsections describe the steering schemes along with several evaluations that motivate their design. A more complete evaluation including comparison with other previously proposed schemes in the literature will be conducted in section 4.3.

4.2.2 Modulo Steering

This is actually not a dependence-based scheme, but we introduce it here for comparison purposes. This is a naive scheme that sends alternatively one instruction to each cluster. It is simple and very effective regarding workload balance, but may result in a high communication overhead because there is no consideration on communication.

4.2.3 Simple RMB Steering

The first RMB steering algorithm we have considered is the Simple RMB scheme. First, it selects the clusters where all or most of the source operands are mapped, as described above, then it chooses one of them randomly. The algorithm works as follows:

1. Select clusters with highest number of source registers mapped (as described in section 4.2)
2. Choose one of the above selected clusters, randomly

A communication is originated just in the case when one instruction has two source registers and there is not any cluster that has both registers mapped. No consideration of balance is taken, relying on the fact that the random steering used when the communication overhead is the same for several clusters is good enough to keep the balance steady.

4.2.4 Balanced RMB Steering

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

1. Select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

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

Figure 4-1 shows the IPC for the Modulo, Simple RMB, Balanced RMB, and the upper bound centralized architecture. Figure 4-2 shows the average number of communications per committed instruction, and figure 4-3 shows the average workload imbalance. As expected, the modulo scheme is the best balanced one, since it sends one instruction to each cluster alternatively. However, while this scheme achieves a near optimal workload balance, it shows the worst performance, because the advantage in balance cannot offset the overhead produced by a huge amount of inter-cluster communications (almost 98% of the instructions executed require communications).

In contrast, the Simple RMB and the Balanced RMB schemes perform significantly better due to their much lower communication overhead. The Balanced RMB steering has a better

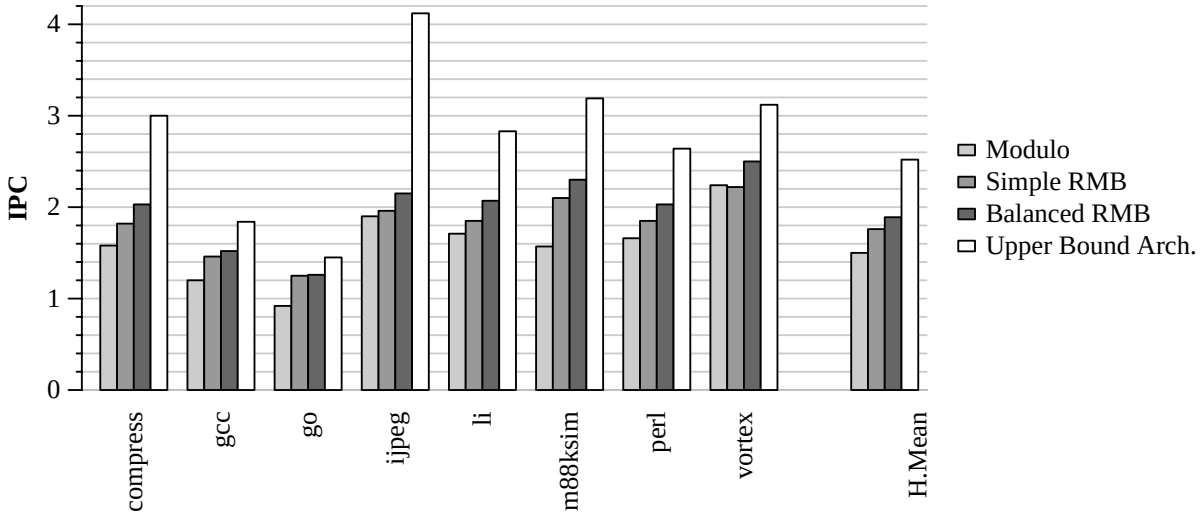


Figure 4-1: Performance of Modulo, Simple RMB and Balanced RMB steering schemes on a four-cluster architecture

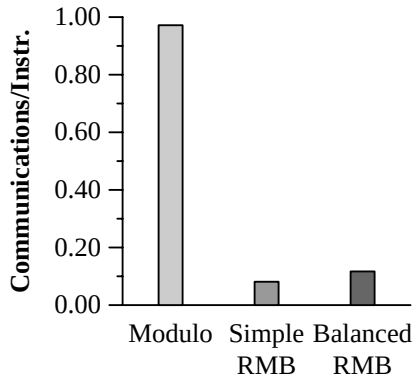


Figure 4-2: Average number of communications per dynamic instruction

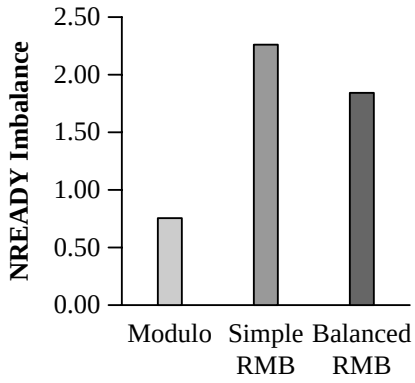


Figure 4-3: Average NREADY workload imbalance

workload balance than the Simple RMB, as shown in figure 4-3, due to the balance considerations it implements, and therefore, it achieves also a better performance.

Overall, the above results show that, while it is important to achieve a good workload balance among the clusters, it is also important to reduce the communications. We can conclude that, among the three algorithms, the Balanced RMB scheme performs the best because it achieves the best trade-off between communication and workload balance.

However, the IPC of the Balanced RMB is still significantly below that of the centralized - upper bound - architecture (figure 4-1). To attempt improving it we have also studied the behavior of the workload balance along the execution of each program. We measured the distribution function of the NREADY metric for each program. This function is depicted in Figure 4-4, only for a particular benchmark (perl) although we found that the shape of the distribution is quite similar for all the benchmarks examined. The graph shows that the most frequent imbalance value is zero, but this is not a surprising result, since the NREADY metric only considers imbalance as the situation where some cluster has more ready instructions than

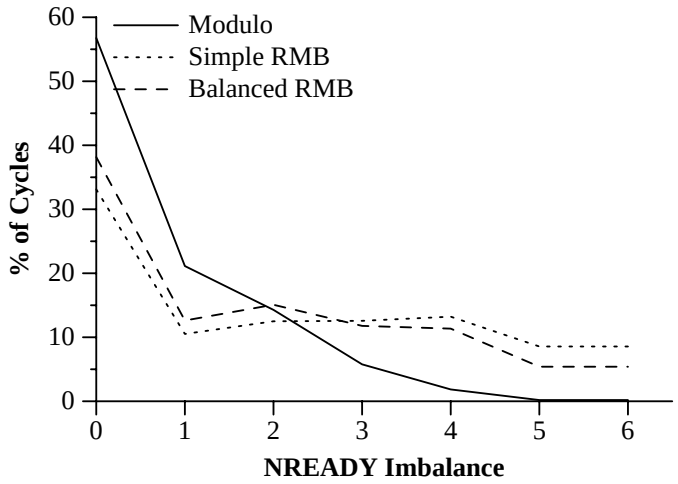


Figure 4-4: Distribution function of the NREADY workload imbalance (perl)

the issue width and another cluster has fewer. The most remarkable feature of this graph is that high imbalance values (NREADY > 2) are not infrequent, which motivates the proposal of a new steering scheme.

4.2.5 Improving the Workload Balance with the Advanced RMB Steering

Figure 4-1 showed that the harmonic mean IPC of the Balanced RMB is 1.89, still below that of the centralized architecture. Since Figure 4-4 showed that high imbalances (NREADY>2) are not infrequent, which suggests that there is still some potential improvement by trying to avoid those strongly imbalanced situations. The Advanced RMB scheme is similar to the Balanced RMB, with a higher emphasis in the workload balance. This scheme checks whether the imbalance has exceeded a given threshold and in this case it ignores the primary criterion, and it just sends the instruction to the cluster that most favours the balance. The algorithm works as follows:

1. Select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

Exception: if imbalance is greater than a given threshold, then ignore rule 1

Of course, this approach may decide that an instruction executes in a cluster where none of its operands are mapped, due to the poor workload balance at that moment, and therefore it may increase the number of inter-cluster communications. The threshold determines a trade-off between workload balance and inter-cluster communications. If the threshold is set too high, then we get little improvements on workload balance, but if it is set too low, then the communication overhead offsets the balance improvements. Some experiments have been conducted in order to find out the best threshold that triggers re-balancing. We found the best values to be DCOUNT=16 for 2 clusters, and DCOUNT=32, for 4 clusters.

In Figure 4-5 it is shown the IPC for the Modulo, Balanced RMB and Advanced RMB. Figure 4-6 shows the average number of communications per executed instruction, and figure 4-7 shows the average workload imbalance. It is shown in figure 4-7 that the Advanced RMB scheme achieves the best workload balance. This workload balance improvement is better

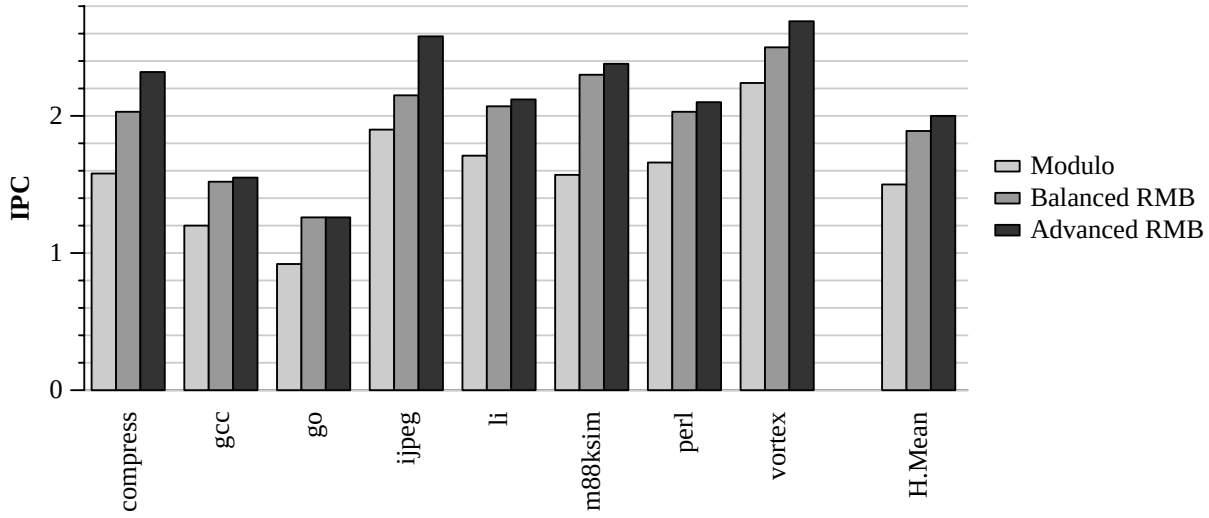


Figure 4-5: IPC of the Advanced RMB vs. Balanced RMB steering schemes on a four-cluster architecture

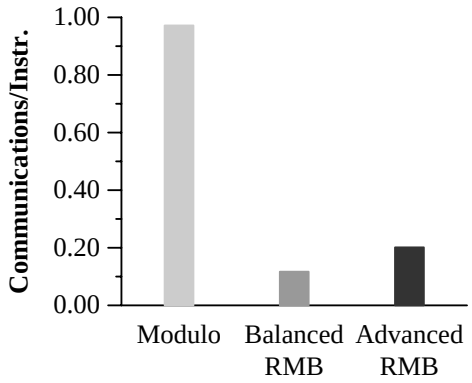


Figure 4-6: Average number of communications per dynamic instruction

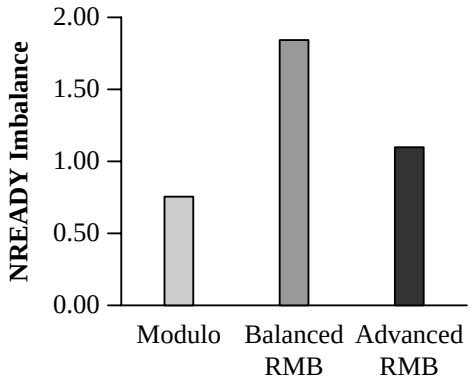


Figure 4-7: Average NREADY workload imbalance

explained in figure 4-8, which depicts its distribution function. It shows that the rebalancing actions produced by the Advanced RMB steering scheme mostly tend to reduce the frequency of situations with a high NREADY imbalance. As shown in figure 4-6, such a balance improvement comes at the cost of an increase in communications because, during rebalancing actions, the dependence criterion is ignored. However, as shown in figure 4-5, the Advanced RMB outperforms the other schemes, because it achieves the best trade-off between communications and workload balance.

4.2.6 Optimizing the Critical Path with the Priority RMB Steering Scheme

One of the main goals of a steering algorithm is to minimize the performance penalty associated to inter-cluster communications. However, not all communication delays result in a loss of performance but only those among instructions in the critical path of execution. Therefore, a more effective approach would be to minimize the number of critical communications.

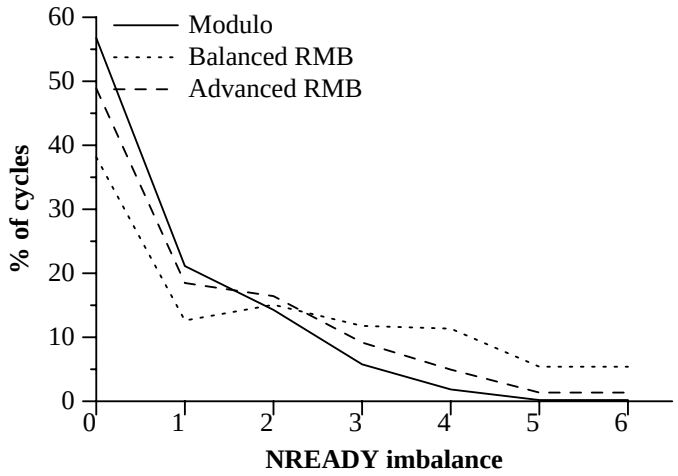


Figure 4-8: Distribution function of the NREADY workload imbalance (perl)

This kind of considerations motivate our Priority RMB scheme, which addresses the relative criticality of the communications within the scope of a single instruction. Note that among the true register dependences of an instruction, only the last operand being produced may be critical. Our approach is based on the observation that if an instruction has two source registers, and only one of them is available, the unavailable operand is more likely to be critical, because it will be produced later. Consequently, the Priority RMB steering scheme is similar to the Advanced RMB, except that it considers the dependences through unavailable operands prior to other dependences. The algorithm works as follows:

1. To minimize communication penalties:
   1.1. If there is any unavailable operand, choose its producer cluster
   1.2. Else, select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

Exception: if imbalance is greater than a given threshold, then ignore rule 1

Of course, this scheme requires that the steering logic monitors the availability of the operands at the time it distributes instructions. It is possible for some implementation that moving register availability information from the issue logic to the front-end cannot be done instantly, but after some delay. In that case, the steering logic will operate with partially outdated register availability information. The only effect of this is that, for a very small number of instructions, the steering will wrongly give priority to one of its source operands, but it will never affect execution correctness. We found that steering with a 1-cycle outdated availability information produces negligible performance effects for most benchmarks (0.8% IPC on average).

Figure 4-9 shows the performance of the Priority RMB compared to the Advanced RMB scheme. The Priority RMB scheme performs slightly better than the Advanced RMB for some of the benchmarks, because it reduces the latency of fetching critical operands, although the average performance improvement is small (0.2% IPC).

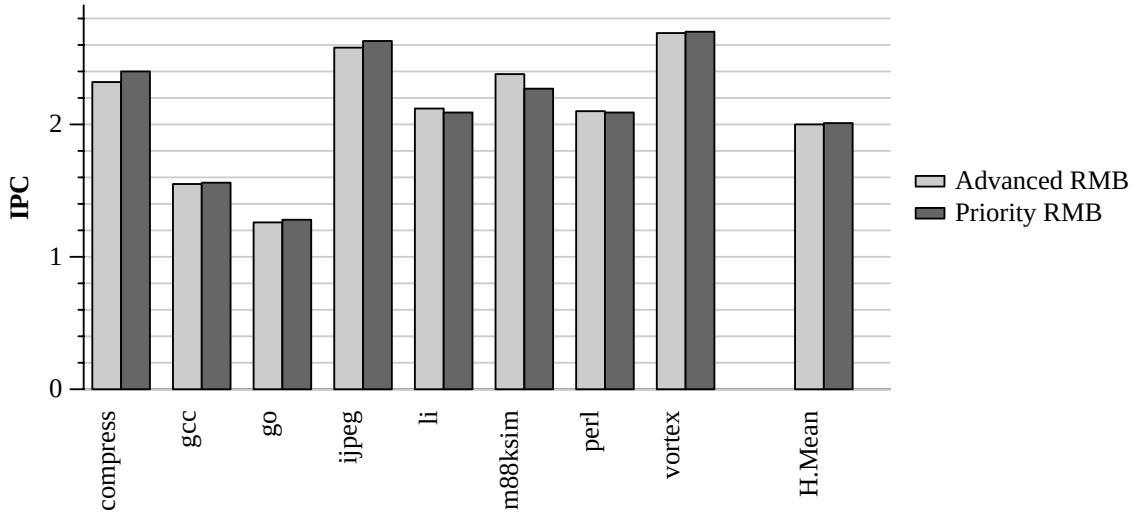


Figure 4-9: IPC of the Priority RMB vs. Advanced RMB steering schemes on a four-cluster architecture

Another approach for giving priority to critical operands could be to predict which operand will be produced last, based on previous program behavior [11]. This question opens a new research direction that has been left for future work.

4.2.7 Reducing Communications with Accurate-Rebalancing

One major drawback of the previous cluster assignment algorithm is that it generates too many communications during the periods when the workload imbalance exceeds the threshold, because then it totally ignores dependences. Moreover, the probability that the steering algorithm generates a communication in such cases grows with the number of clusters. We observed that most often, a strong imbalance situation is caused by a single overloaded cluster. Of course, rebalancing the workload does require to steer instructions to the less loaded clusters, but choosing strictly the least loaded one is probably not a critical factor. Of course, for a two-clustered organization there is no other alternative, but for four or more clusters the steering scheme could recover the strong imbalance with more accurate rebalancing actions that do not ignore completely the dependences, thus not generating as many communications.

We propose that, in case of a strong imbalance situation, instead of directly choosing the least loaded cluster, the algorithm follows the normal criteria except that the most loaded clusters are previously excluded from the choice of clusters. In doing so, there is a chance that among the non excluded clusters there is one where the source registers are mapped and thus inter-cluster communications are not required. We experimented with different exclusion criteria based on the existing signed workload counters (refer to section 4.1.2 for details), and found the most simple and effective is to exclude clusters that have a positive workload counter. Note that the Accurate Rebalancing (AR) technique could equally apply to the ARMB and the PRMB schemes proposed in this section. For the experiments in this chapter, it was assumed only the AR-PRMB scheme.

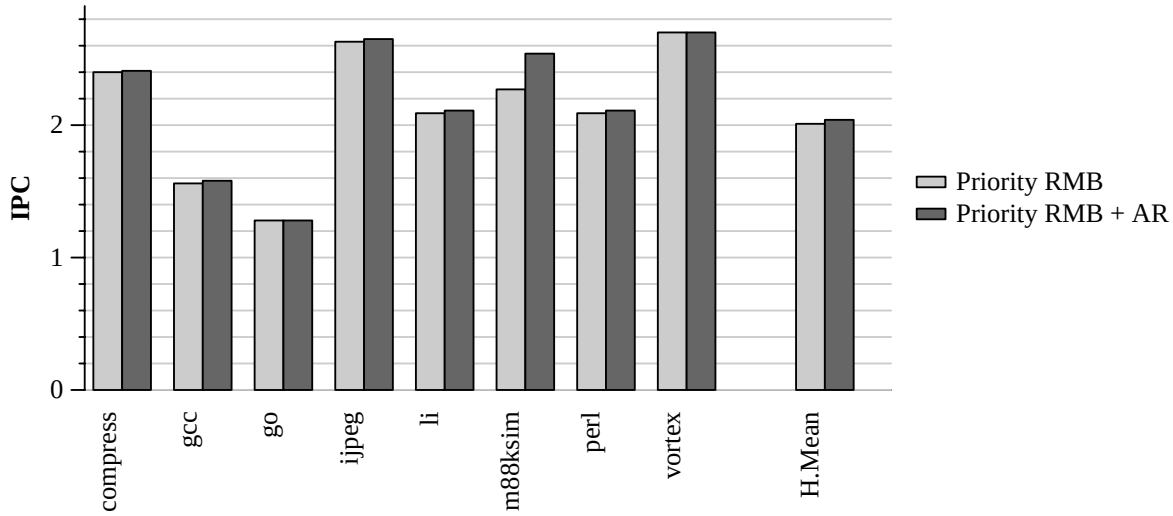


Figure 4-10: IPC of the Priority RMB steering scheme with Accurate Rebalancing, and without it, for four clusters

In more detail, this scheme works as follows:

1. To minimize communication penalties:
   1.1. If there is any unavailable operand, choose its producer cluster
   1.2. Else, select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

Exception: If imbalance > threshold, exclude clusters with workload 0, prior to applying rules 1, 2

Figure 4-10 shows the performance of the Priority RMB with Accurate Rebalancing and without it. It shows that the Accurate Rebalancing improves performance by 1.8% on average, because it reduces communications (from 0.23 to 0.20 communications per instruction, on average).

4.3 Evaluation

In this section, the RMB steering schemes are compared to the best steering schemes proposed in the literature. Next, we study the sensitivity of the various schemes to the latency and bandwidth of the interconnect. Finally, to give our conclusions a higher generality, the proposed RMB steering schemes are evaluated assuming two-cluster and four-cluster architectures, and the experiments use not only the SpecInt95 but also the Mediabench benchmark suites.

4.3.1 RMB versus Slice-Based Steering Schemes

A performance comparison of the Advanced RMB scheme against the slice-based schemes was already reported in the previous chapter (see figure 3-15), since the scheme referred to as the general balance actually matches the definition of the Advanced RMB, as already noted in section 3.3.8.

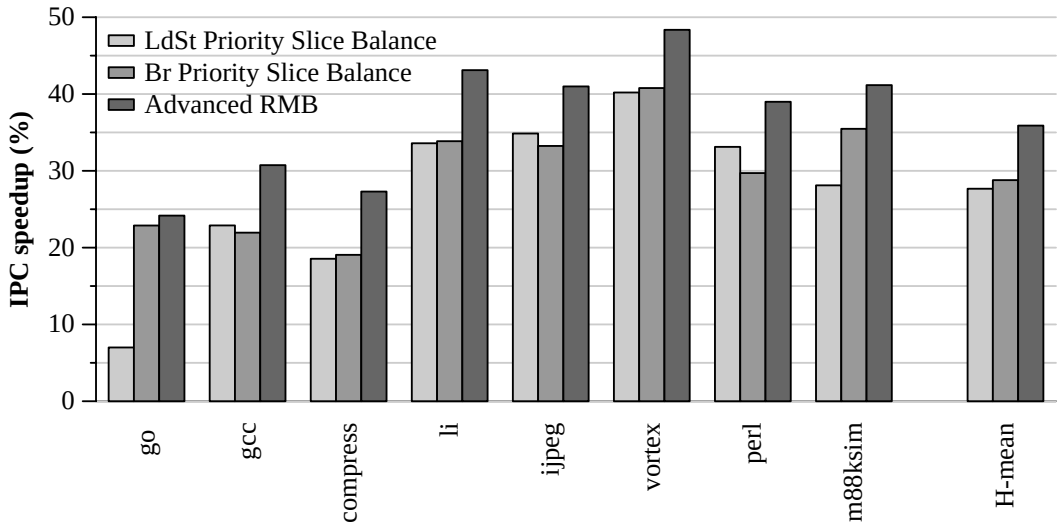


Figure 4-11: The Advanced RMB versus the best slice-based steering schemes, on a cost-effective architecture (IPC speedups over a conventional superscalar)

Figure 4-11 compares the performance of the Advanced RMB scheme (taken from figure 3-14) against that of the best slice-based scheme, i.e. the Priority Slice Balance (taken from figure 3-13), Performance is reported as the speed-up of the cost-effective two-clustered architecture over a conventional superscalar with similar complexity (see table 3-2). The results show that the Advance RMB scheme outperforms all the proposed slice-based steering schemes and achieves an average speedup of 36%. This result shows that instruction-based schemes are more efficient than slice-based ones because they work at a finer granularity, which makes them more adaptable.

4.3.2 The AR-Priority RMB versus Other Instruction-Based Schemes

We compare the Accurate Rebalancing Priority RMB scheme (AR-PRMB) to the best dynamic instruction-based approaches in the literature. It is compared to the FIFO-based scheme proposed by Palacharla, Jouppi and Smith [70,71], and also to the MOD3 scheme proposed by Baniasadi and Moshovos [10].

The basic idea of the FIFO-based scheme is to model each issue queue as if it was a collection of FIFO queues. Instructions are steered to FIFOs following an heuristic that ensures that any two consecutive instructions in a FIFO are always dependent. If all the source registers are ready, or their producers do not stay at any of the FIFO tails, then the instruction is dispatched to an empty FIFO. The assignment policy chooses always empty FIFOs from the same cluster until all of them are busy, then it switches to another cluster in round-robin order (for more details refer to the original paper [70]). In our experiments, we modelled the FIFO-based steering with 8 FIFOs per cluster, each with 4 entries (thus having twice the number of scheduler entries per cluster as our model).

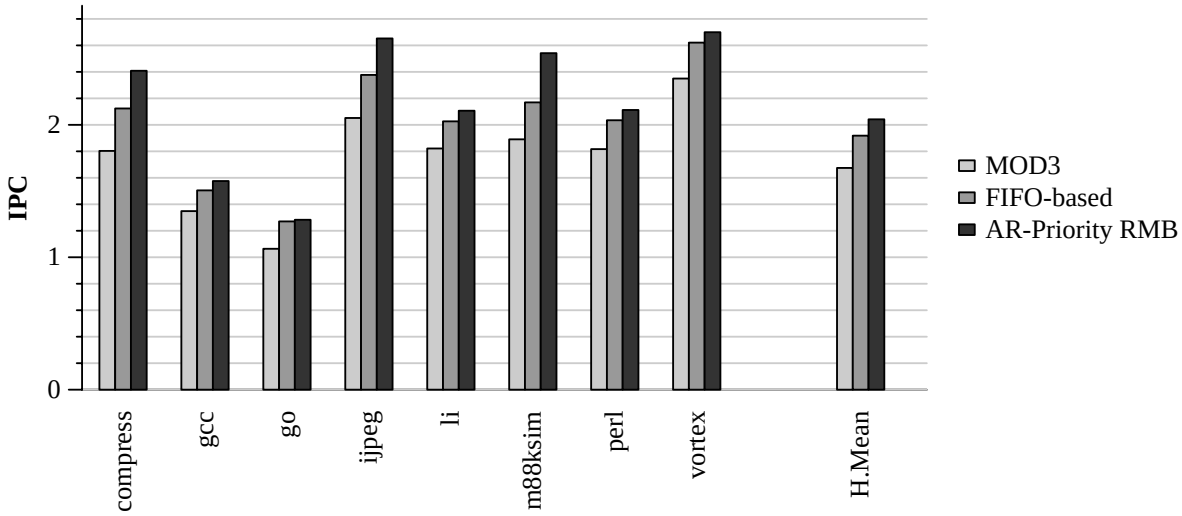


Figure 4-12: Performance of the AR-Priority RMB, FIFO-based and MOD3 steering schemes, for four clusters

Figure 4-12 shows that the AR-PRMB scheme significantly outperforms the FIFO-based steering scheme for all the programs, with an average improvement of 6.4%. The main reason for such a difference is that the FIFO-based approach generates 0.26 inter-cluster communications per dynamic instruction while the AR-PRMB generates only 0.20. Note that the FIFO-based scheme cannot drive an instruction to the cluster that produces its operands if they are ready or if the producer does not stay in a FIFO tail because another dependent instruction was previously steered.

The MOD3 cluster assignment scheme of Baniasadi and Moshovos [10] sends every group of three consecutive dynamic instructions to the same cluster, and then it switches to another cluster following a round-robin policy. Such an assignment does a very good job at keeping the workload balanced. However, it generates a much higher communication rate (0.75 communications per instruction), because it lacks a specific policy for grouping dependent instructions together. Figure 4-12 shows that our AR-PRMB scheme outperforms the MOD3 scheme by 22%.

Baniasadi and Moshovos [10] found that the MOD3 steering scheme outperforms a dependence-based scheme. Our conclusion differs from theirs, because we assume a different architecture. First, because our model assumes an additional 1-cycle overhead for issuing the copy instructions, which stresses the importance of reducing communications. Second, because they assumed a replicated register file (like the one in the Alpha 21264 [53], and the one assumed by Palacharla [70]). With such a register file, the steering logic is not concerned with dependences through ready source operands because they may have been produced well in advance, and they are probably available in all clusters. In contrast, with a distributed register file, the steering scheme must take them into account (e.g. using renaming information like in our RMB schemes) to avoid costly communication overheads.

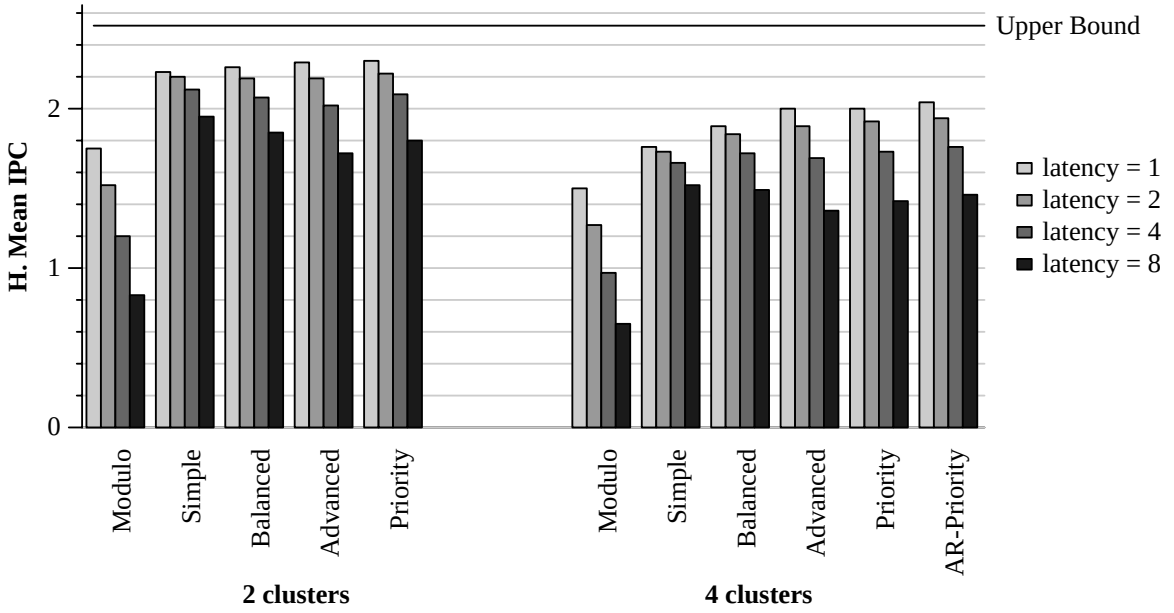


Figure 4-13: Performance sensitivity to the communication latency

4.3.3 Sensitivity to the Communication Latency

In future architectures, inter-cluster communication latency will likely increase because the interconnect wire length increases with processor complexity, and because of the widening gap between the relative speeds of gates and wires on future technologies. Using high clock rates will require not only to reduce the capacity of many components like register files and issue windows, but also to pipeline more deeply the access to other structures.

In previous sections we have assumed that inter-cluster communications take 1 cycle (there is a 1 cycle “bubble” between the copy instruction and the dependent instruction, in a different cluster). In this section, we study the sensitivity of clustered architectures to the communication latency, measured by the IPC degradation caused by a communication latency of 1, 2, 4, and 8 cycles. In all cases we assume that communications are fully pipelined, that is, for a given bypass path, one communication may begin per cycle regardless of its total latency.

Figure 4-13 shows the performance of the steering schemes presented in section 4.2 for a range of communication latencies between 1 and 8 cycles, with two and four clusters. Of course, the results for a 1 cycle latency and 4 clusters are the same as those presented in section 4.2.

As expected, the performance loss is higher with four clusters than two, due to the higher communication rate. We can also observe that, as latency increases, there is a growing performance loss and this loss is much greater for the modulo scheme than for the others, since it does not take into account instruction dependences. This behavior stresses the increasing importance of minimizing the number of communications. We can also see significant differences among the five RMB schemes: for small latencies, the AR-Priority, the Priority and the Advanced schemes outperform the other two because they achieve a better trade-off between

communications and workload balance. However, as latency increases beyond 4 cycles, they perform worse because the performance loss is dominated by the communication penalty, and it is no longer worth trading communication for workload balance.

For high latencies, the performance of all the analyzed schemes is quite below the upper bound of the centralized architecture. In this scenario, the steering schemes should make more emphasis in reducing the number of communications, even at the expense of a worse workload balance. Alternatively, other communication saving techniques could be implemented, such as those proposed in the next chapter.

4.3.4 Sensitivity to the Interconnect Bandwidth

The inter-cluster communication bandwidth has a direct impact on the complexity and delay of the register files [28] and the bypass network, since it determines the number of register file write ports devoted to remote accesses, the number of bypass multiplexer's inputs coming from remote clusters, and the number of outputs from the bypass network to the interconnection network. Furthermore, the inter-cluster communication bandwidth also determines the number of tags that are broadcast to the instruction queues of remote clusters. Therefore, it has a direct impact on the complexity and delay of the wake-up logic, which depends quadratically on the total number of tags crossing its CAM cells [70, 71].

So far, we have assumed an unbounded bandwidth for the interconnection network to isolate our results from possible communication bandwidth bottlenecks. Here we study the negative impact of contention delays when assuming a limited bandwidth. For an N-cluster configuration, we assume a simplified model with NxB independent paths. Each path is implemented through a pipelined bus where any cluster can send a value and each bus is connected to the write port of a single cluster register file. Therefore, we assume that each register file has B write ports for inter-cluster communications. Any cluster may allocate one of these paths to write a value to a remote register file, and holds it during a single cycle, since the communication is fully pipelined. Obviously, this model is somewhat idealized, since it omits the complexities due to pipelining, arbitration, or variable latencies dependent on the topology, but it may provide a first order approach to evaluate the problem.

We evaluated the performance sensitivity to the interconnect bandwidth, within a range of 1 to 4 incoming message per cluster each cycle and also with an unlimited bandwidth on a four-clustered architecture. The results in Figure 4-14 correspond to the AR-Priority RMB steering scheme, and they show that the performance degradation caused by contention delays when the communication bandwidth is reduced to a single incoming path per cluster is almost negligible. Although not shown, similar results were observed with the rest of RMB schemes. This result is consistent with the low number of communications per instruction of the RMB schemes, as presented in previous sections (and also summarized in figure 4-16 below). We also found that only the modulo steering achieves a significant performance improvement as bandwidth increases, because of its huge communication demands (see figure 4-2).

In consequence, these results suggest that for inter-cluster communications on a clustered architecture with an RMB steering scheme it may suffice just a single write port in each register

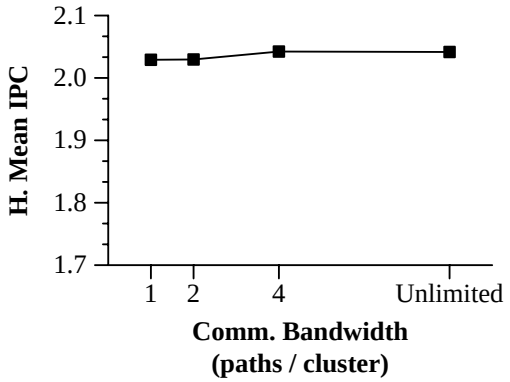


Figure 4-14: Sensitivity to the interconnect bandwidth for four clusters (AR-Priority RMB steering)

file, a single incoming tag per issue window, and a single remote bypass attached to the input multiplexers of the functional units. Moreover, the interconnect can be implemented with low hardware cost and complexity. Several proposals and a more in-depth study of the interconnect network will be conducted in Chapter 6.

4.3.5 Overall Evaluation of the RMB Steering Schemes

In section 4.2, performance results were reported individually for each RMB steering scheme Here, all of them are gathered in a single graph to allow a direct comparison, and the scope of the experimental assumptions is extended, for the sake of a higher generality. Two eight-way issue superscalar architectures are studied, with two and four clusters respectively (see parameters in table 2-1). In addition, the experiments shown below use both the SpecInt95 and the Mediabench benchmark suites (see details in section 2.3).

Figure 4-15 compares the harmonic mean IPCs for the six steering schemes. Figure 4-16 compares the average number of communications per instruction, and figure 4-17 compares the average NREADY workload imbalance metric. In all cases, the best performing heuristics is the AR-Priority RMB scheme. The results with two clusters show similar trends as those with four clusters, although the differences among the various steering schemes are smaller, because of the lower communications and workload imbalance overheads (figures 4-16 and 4-17).

The results with the Mediabench benchmark suite show similar trends as those studied with the SpecInt95, but the differences among the various steering schemes are more prominent (the AR-Priority RMB scheme for four clusters outperforms the Simple RMB scheme by 32%, while it was only 16% with the SpecInt95). The Mediabench programs have more ILP than the SpecInt95, which make them more prone to suffer high workload imbalances. Therefore, improving the workload balancing ability of the steering also has a greater positive impact on performance.

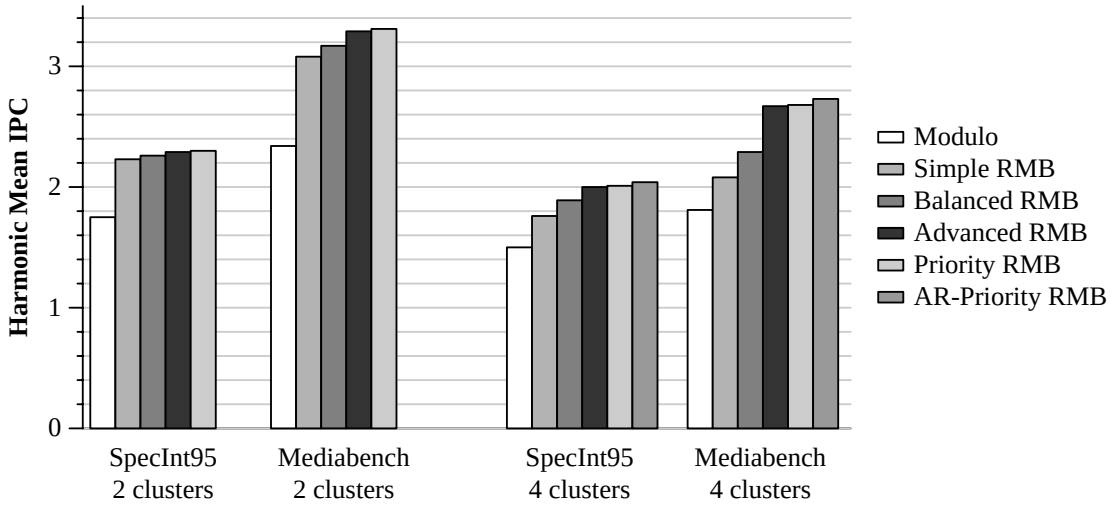


Figure 4-15: Overall performance of the RMB steering schemes

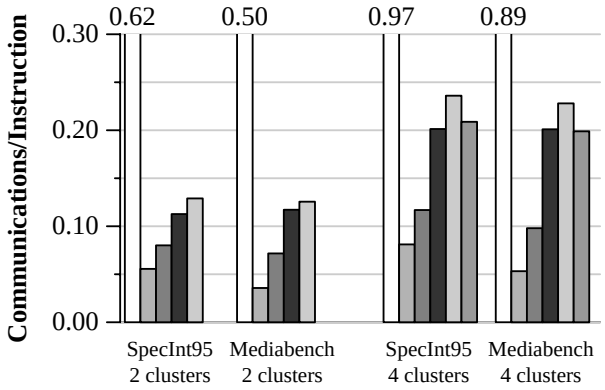


Figure 4-16: Average number of communications per instruction

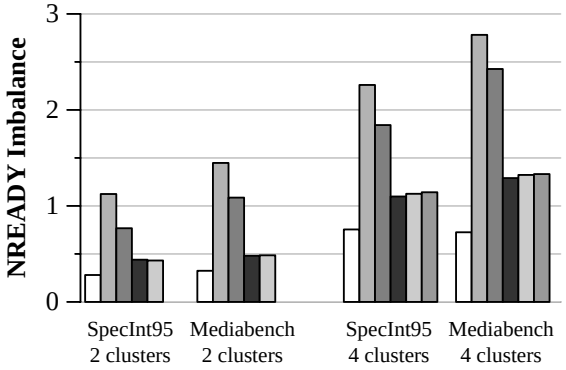


Figure 4-17: Average NREADY workload imbalance

4.4 Conclusions

We have proposed four new instruction-based mechanisms that dynamically partition a sequential program into the different clusters of a clustered microarchitecture. Instruction-based schemes perform finer-grain cluster assignments than slice-based ones, on a per-instruction basis, so they are more flexible and more effective than slice-based schemes. The proposed RMB schemes follow a primary and secondary criteria that specifically address the goals of minimizing communications and maximizing workload balance respectively.

We have proposed a new workload balance metric (NREADY) that matches the intuitive concept of workload balance and, unlike previous proposals, it is suitable for any number of clusters. Likewise, we propose a new imbalance counter mechanism, suitable for any number of clusters (DCOUNT), which is used to guide efficiently the steering decisions of the proposed RMB schemes.

Our experiments show that the proposed AR-Priority RMB steering scheme is more effective than all the existing slice-based schemes, either static or dynamic, and it significantly outperforms the best dynamic schemes previously proposed in the literature, because it achieves the best trade-off between communications and workload balance.

Finally, we found that a clustered architecture is quite sensitive to the inter-cluster communication latency, which emphasizes the importance of reducing communications with appropriate steering decisions. However, we found that, with our RMB steering schemes, it is hardly sensitive to the cluster interconnect bandwidth, because it has a low bandwidth demand. Therefore, inter-cluster communications can be implemented with very low complexity impact on the register files, the issue windows, the bypasses, and the cluster interconnect networks, which results in short cycle times and low power consumption.

CHAPTER 5

REDUCING WIRE DELAY PENALTY THROUGH VALUE PREDICTION

In this chapter we show how value prediction can be used to avoid the penalty of long wire delays by providing the receiver of the communication with a predicted value and validating the prediction locally, where the value is produced. Only in the case of misprediction, the long wire delay is experienced. This concept is applied to a clustered microarchitecture in order to reduce inter-cluster communications. In addition, the predictability of values allows the dynamic instruction partitioning hardware to have less constraints to optimize the trade-off between communication requirements and workload balance, which is the most critical issue of the partitioning scheme. In particular, we show that the performance of a realistic implementation of a four-cluster architecture may be improved by 14% through a simple value prediction scheme and a new steering logic designed to take advantage of the value predictor.

5.1 Introduction

As discussed in Chapter 1, the increasing impact of global wire delays will become soon a major problem in the design of future microarchitectures because signals that cross a large portion of the die will require multiple cycles to propagate. By predicting the value to communicate, the receiver may proceed without being penalized by the communication delay. The actual communication may occur later on, out of the critical path of execution, or it may be replaced by a simple verification signal, depending on where the verification takes place.

Value prediction has been largely investigated in the context of superscalar processors, and it is not our purpose to design another predictor but to investigate its potential to reduce the penalties of slow communications. Value prediction has a great potential to eliminate data

dependences and to increase program parallelism, but it sometimes does not fulfill the expectations because the misprediction overhead offsets almost completely the performance gains. Therefore, in this chapter we propose to revisit the value prediction technique in the context of long communication latencies caused by wire delays.

As a sample application, we will show how value prediction helps reducing inter-cluster communication penalties in a clustered architecture, and enables a new source of performance improvements. In conventional value prediction approaches, instructions are dispatched speculatively with a predicted source register only if its actual value is still unavailable. Our proposal differs from this because an instruction may be dispatched speculatively with a predicted source register even though the computed value is already available, if this value is in a remote cluster.

In addition, we will show how value prediction significantly improves the effectiveness of the steering logic by providing a less dense data dependence graph which results in less communication requirements and better opportunities to balance the workload. Since both inter-cluster communications and workload imbalance usually produce a high IPC loss, a clustered architecture may benefit from value prediction more than a centralized one.

5.2 Microarchitecture

This section describes the basic value prediction mechanism assumed in this chapter, as well as the required extensions that we propose for a clustered architecture, and presents a performance evaluation.

5.2.1 Value Prediction

We assume that the microarchitecture implements a stride value predictor [25, 37, 38, 88] that predicts the source operands of the instructions. It has a tagless value prediction table indexed by the PC and by the operand order (left/right). We first assume a very large table (64K entries) to isolate the results from the effects of a limited table size, and we later evaluate the impact of a table with sizes ranging from 1K to 64K entries (section 5.4.3). Each entry contains the last value, the last observed stride and a 2-bit counter that assigns confidence to the prediction. On a misprediction, the stride is updated, but only if the confidence counter is lower than 3. Such an updating policy [39] avoids mispredicting twice on many inner loops, and has a similar purpose and performance as the 2-delta stride technique [25]. Since each prediction involves a table access and an addition, we assume that value predictions are available 1 cycle after the fetch, i.e. at the decode stage. Table updates are done at decode time.

When a source operand is not yet available at dispatch time, and its predicted value is confident (the confidence counter is greater than 1), the instruction is dispatched speculatively and may use the predicted value. The instruction that will produce this value is identified, and it is assigned the task of verifying that its output matches the prediction. The verification occurs

during the writeback stage of the producer instruction, and it takes one cycle. If it fails, the dependent misspeculated instruction is invalidated and reissued.

We have assumed a selective invalidation and reissue mechanism [57, 82], i.e. after the mispredicted instruction is reissued and executed, a new value is produced and propagated to dependent instructions, which in turn reissue, and so on. Only the instructions that depend on the mispredicted instruction are invalidated. The mechanism is in fact the existing issue mechanism, and therefore we have assumed no additional penalty for each instruction restart.

Since mispredictions are found late in the pipeline, during the writeback stage of the producer instruction, the misspeculated dependent instructions are actually re-issued several cycles later than they would do if they were not speculative (see diagram in figure 5-6). Hence, they are effectively delayed by as many cycles as pipeline stages between issue and writeback. Even though speculation is restricted by the confidence bits to the most predictable values, the penalty incurred by mispredictions may still be so high that it offsets most of the performance gains of correct predictions. Improving the predictor accuracy and restricting speculation to those instructions with a higher impact on the critical path length [31] are valid approaches to reduce these overheads. It is beyond the scope of this thesis to study these alternatives, however we found that speculating only on values that are the results of loads (many of which are likely in the critical path, due to cache misses) provides an additional 2.7% average performance improvement, so this simple constraint is assumed for the rest of the experiments in this chapter.

5.2.2 Speculation on Remote Operands

For a clustered architecture, the above speculation procedure is further extended, in order to reduce inter-cluster communications, which is a major goal of our proposal. The extension applies to the case when a source register is not currently mapped on the cluster where the instruction is being dispatched. In this case, the instruction is dispatched speculatively with the predicted value, even if the register is unavailable, and a special *verification-copy* instruction is dispatched to the cluster where the operand is to be produced, instead of a normal copy instruction. During the cycle following the read stage, the verification-copy compares locally the prediction with the computed register value, and it sends the corresponding validation signal through the interconnect. The actual communication of the correct value is only required in case of comparison mismatch, and then the remote misspeculated instructions must re-issue with the correct input.

5.2.3 The Baseline Steering Algorithm

The cluster assignment algorithm assumed for our baseline clustered architecture is the Priority RMB scheme (PRMB), since it was shown to be the most effective (see chapter 4). In normal operation, this algorithm firstly selects from among clusters to minimize communication penalties and secondly, it chooses the least loaded cluster. However, when the workload imbalance exceeds a given threshold, then the first criterion is ignored, and the least loaded cluster is always chosen. We reproduce the algorithm here for clarity:

1. To minimize communication penalties:
   1.1. If there is any unavailable operand, choose its producer cluster
   1.2. Else, select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

Except: if imbalance is greater than a given threshold, then ignore rule 1

5.2.4 Performance Evaluation

We ran a set of experiments to compare the performance impact of value prediction on a clustered architecture, with two and four clusters, and also on a centralized architecture, all of them with similar resources and identical pipeline length (see table 2-1 for details). These experiments use the Mediabench benchmark suite [56, 62] and assume all the experimental setup described in section 2.2, including the simulator and the architectural parameters as well as the stride value prediction and the PRMB steering scheme described above.

Figure 5-1 shows the IPC of these three architectures without value prediction, which will be referred to as the baseline architectures. These baseline IPC results will be used for further speedup results in this chapter. Figure 5-2 shows the speedups achieved when value prediction is used in each of the three architectures. These results show that the impact of value prediction on a centralized architecture is very small, just a 2.9% speedup on average, and negative for several benchmarks. In contrast, the average speedup on a four-clustered architecture is 9.5% (5.5% for two clusters), because value prediction drastically reduces inter-cluster communications per instruction from 0.22 to 0.13 on average (from 0.12 to 0.08, for two clusters). There is one exception, *rawdaudio*, which loses 4% IPC on a 2-cluster architecture due to a slight increase of communications.

5.3 A Steering Scheme for Value Prediction

In this section we focus on how the steering logic of a clustered processor may improve its effectiveness by being aware of the existing value prediction mechanism. Let us assume that predicted source registers will never cause communications or delays, which is true if the prediction does not fail. Then, we could relax the constraints imposed by dependence-based steering criteria (rule 1, in section 5.2.3) for predicted operands, in order to concentrate on improving the workload balance (rule 2). As far as the misprediction rate is kept low, this policy may improve significantly the workload balance.

5.3.1 Enhancing the Partitioning through Value Prediction

In more detail, we propose the following two modifications to the baseline PRMB steering heuristic. First, when the source register of an instruction is predicted and it is not yet available, the steering algorithm considers it as available, thus applying rule 1.2 instead of rule 1.1. In other words, the algorithm does not force to steer the instruction to the cluster where this operand is going to be produced, since it is unlikely that this dependence is in the critical path.

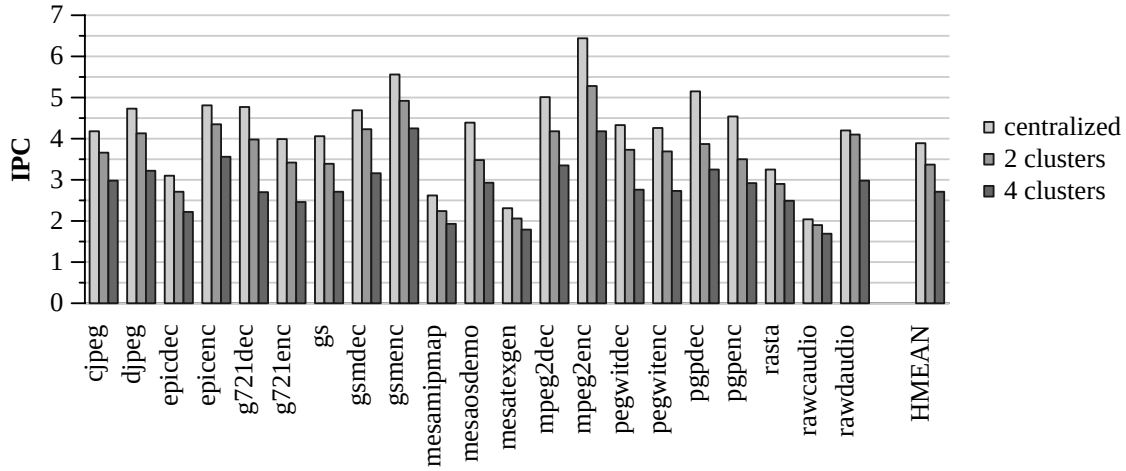


Figure 5-1: IPC of baseline architectures, without value prediction

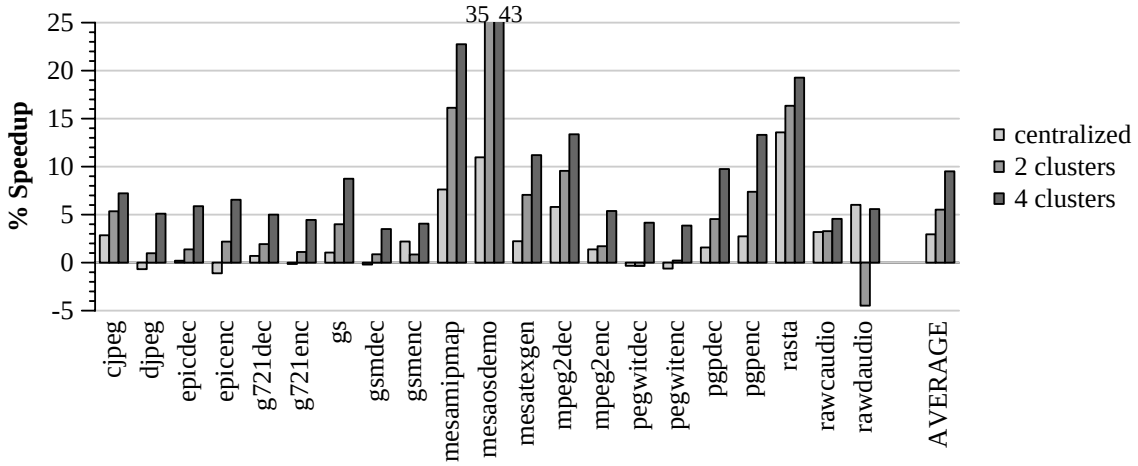


Figure 5-2: Impact of using value prediction on IPC

Second, rule 1.2 of the steering algorithm considers any value-predicted source operand as being mapped in all clusters. As a consequence, this operand does not constrain the set of candidate clusters because, regardless of the cluster it is sent to, it will not cause any additional inter-cluster communication (unless the prediction fails and the operand is remote).

In summary, these two modifications to the steering algorithm eliminate in some cases the constraints imposed by communications/delays issues so that the algorithm has better opportunities for balancing the workload (since rule 2 selects one cluster from a wider choice of clusters).

We evaluated the impact of these two modifications on a four-cluster configuration, and found that they produce worse IPC speedups than the baseline PRMB steering scheme (6.9% average speedup instead of 9.5%). The average NREADY workload imbalance metric (defined in section 4.1.2) is reduced by 32% and, since imbalance correction actions (which ignore communication issues) are less frequent, one would expect also to have less inter-cluster communications. However, the communications ratio (which mostly influences the IPC)

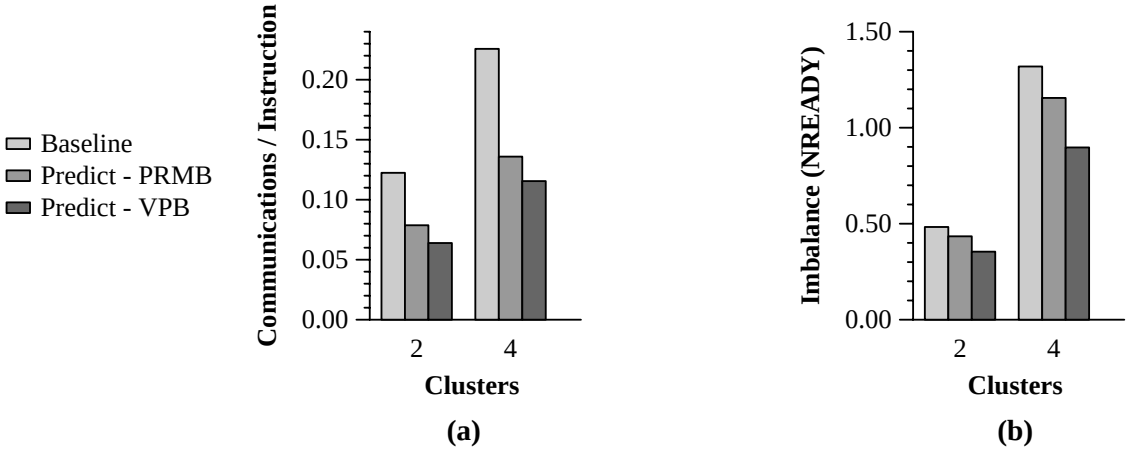


Figure 5-3: (a) Average inter-cluster communications ratio (b) Average workload imbalance

remains almost constant because there is also a communications increase due to an indiscriminate use of the optimistic initial assumptions. More specifically, with the proposed modification to rule 1.2 of the baseline PRMB steering scheme, an instruction that uses a predicted source operand may be steered to a cluster where it is not mapped. In this case, if the prediction fails, the instruction will be re-issued non-speculatively, and a communication will be required to read the correct operand from a remote cluster.

5.3.2 The VPB Steering Scheme

To minimize the above mentioned communications increase, the above mentioned modification to rule 1.2 should only apply to those cases in which there is a potential for improving the workload balance. In particular, we propose that the steering logic considers predicted source registers to be mapped in all clusters only when the DCOUNT workload imbalance counter (see definition in section 4.1.2) is higher than a given threshold. We set this threshold empirically to 16 for four clusters, and 8 for two clusters. In other words, if the workload is very well balanced, the steering does not rely on value prediction to improve workload balance, since it may increase the communication requirements. We refer to this technique as the Value Prediction Based (VPB) steering scheme.

Figure 5-3 compares the average workload imbalance and communication rates for three different cluster architectures: the baseline without value prediction, value prediction with PRMB steering and value prediction with VPB steering, each one configured with either two or four clusters. Figure 5-3 shows the IPC speedups of the two architectures with value prediction over the baseline.

For an architecture with four clusters, it is shown that value prediction with VPB steering requires only 0.11 communications per instruction, which is less than with PRMB steering, and 49% less than the baseline architecture without value prediction. It is also shown that the workload imbalance of VPB steering is lower than that of PRMB steering, and 32% lower than that of the baseline architecture without prediction. Accordingly, while value prediction with PRMB steering achieves only a 9.5% average speedup, with VPB steering it achieves a 14.4%.

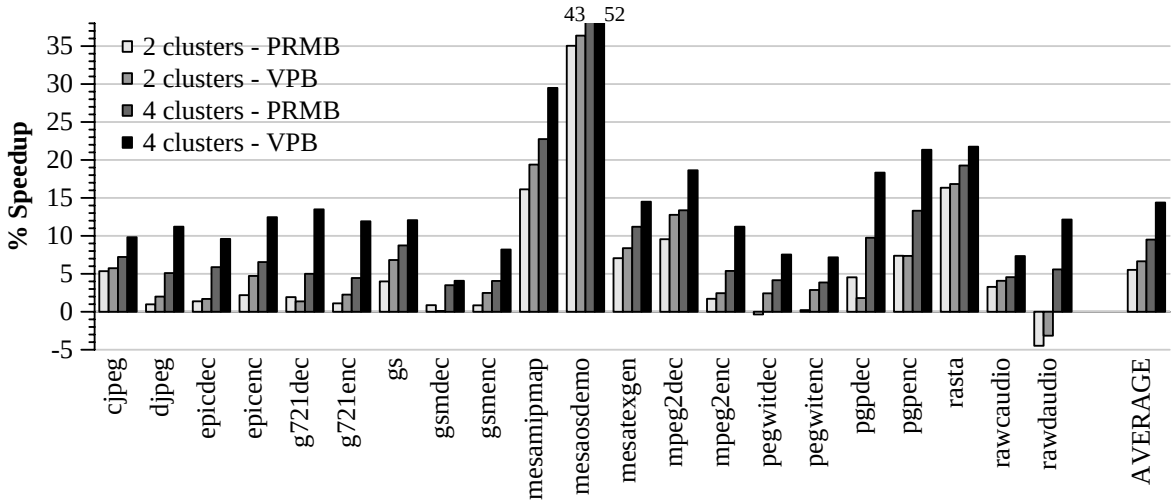


Figure 5-4: Speedups of value prediction, with PRMB and VPB steering schemes

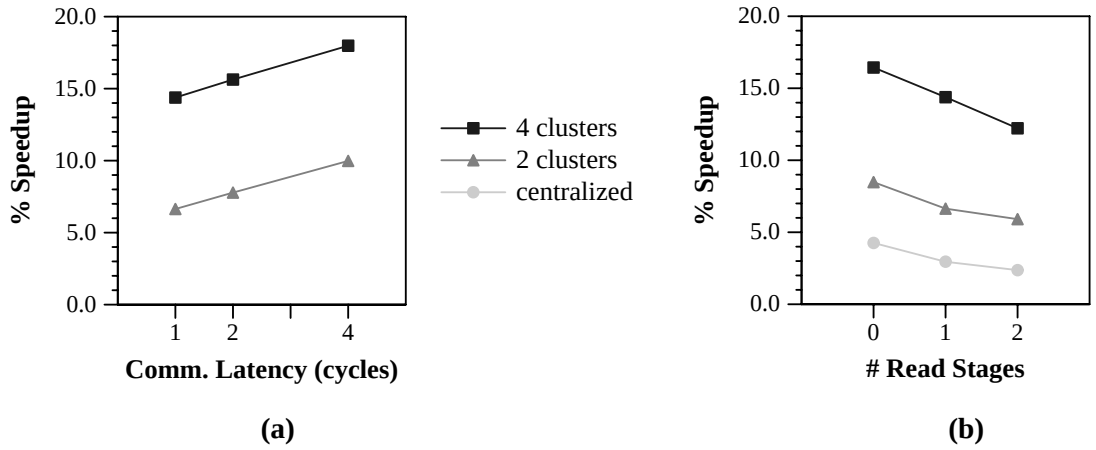
For an architecture with two clusters, the results show similar trends: 0.06 communications per instruction, which is 48% less than the baseline, and a 27% imbalance reduction. However, the performance increase is smaller than for four clusters, a 6.6% average speedup, because there are less communications to be removed with value prediction.

In the above experiments, it was assumed a very conservative value prediction scheme. In order to estimate the potential of this technique with a more accurate predictor, we modelled a perfect predictor and allowed value speculation on any kind of integer values, not only load results. We found speedups of 58%, 38% and 27% for four clusters, two clusters and a single cluster respectively, which suggests that the performance of the VPB steering may significantly be improved by a more effective value predictor, and confirms that the benefit of value prediction increases with the number of clusters.

In summary, we observed that value prediction drastically halves the amount of inter-cluster communications on a cluster organization, and produces significant performance improvements which are higher as there are more clusters, and much higher than for a centralized architecture. We also observed that performance is further increased when adequate steering techniques are implemented that take advantage of value prediction for improving the workload balance. We can thus conclude that value prediction is a very effective technique to reduce the communication requirements of clustered processors.

5.4 Sensitivity Analysis

In this section we analyze the impact on performance of several critical design parameters like inter-cluster communication latency, misprediction penalty and predictor table size.



(a) **(b)**

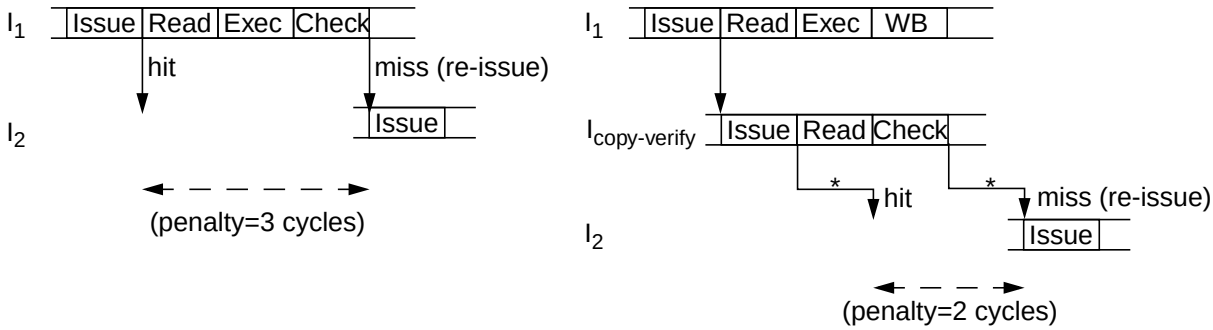
Figure 5-5: Sensitivity of value prediction speedups to (a) communication latency, and (b) latency of the register read stage

5.4.1 Communication Latency

In future technologies the widening gap between the relative speeds of gates and wires will decrease dramatically the percentage of on-chip transistors that a signal can travel in a single clock cycle [1]. Using high clock rates may require not only to reduce the capacity of many components like register files and issue windows, but also to pipeline more deeply the access to other structures, and it may imply a longer inter-cluster communication latency.

In previous sections we assumed that inter-cluster communications take 1 cycle (there is a 1 cycle "bubble" between the copy instruction and the dependent instruction, in another cluster). In section 4.3.3 it was analyzed the impact of the communication latency on performance for several steering schemes, and it was shown how the steering schemes that produced more communications suffer a higher performance degradation. Since value prediction helps reduce the communications demands, one would expect that value prediction also reduces the sensitivity to the communication latency. In other words, the longer the communication penalty, the higher the benefit of eliminating communications. In the following experiments we modelled inter-cluster communication latencies from 1 to 4 cycles, on architectures with 2 and 4 clusters, and measured the speedups of value prediction with VPB steering over the corresponding baseline without prediction.

Figure 5-5a shows that the speedup of value prediction increases with longer inter-cluster communication latencies. When the latency is extended from 1 to 4 cycles, the speedup increases from 14.4% to 18.0% with four clusters (and from 6.6% to 10.0%, for two clusters). Thus, we can conclude that value prediction may become still more useful as wire delays tend to grow with future technologies.



(a) I1, I2 steered to the same cluster

(b) I1, I2 steered to different clusters
(*) inter-cluster latency=1 cycle

Figure 5-6: Timing diagram of two instructions I_1 and I_2, where I_2 mispredicts the value produced by I_1, and must re-issue non-speculatively (the arrows show wakeup signals in case of hit and miss)

5.4.2 Register Read Latency and Misprediction Penalty

As discussed in section 5.2.1, since mispredictions are discovered during the writeback stage of the producer instruction, the mispredicted instruction must re-issue 3 cycles later (2 cycles, for remote operands) than it would do with a correct prediction, as shown in the timing diagram of figure 5-6. As shown, the pipeline depth between the issue and writeback stages determines the minimum misprediction penalty, so it may have a direct impact on IPC. We have modelled several pipelines, having 0, 1 and 2 read stages. The first model does the issue and register read in a single stage (like it occurs in short pipeline layouts), while the other two have 1 and 2 read stages respectively. The first model also corresponds to an architecture that reads the register file before inserting the instruction into the issue queue, like a PowerPC [94].

Figure 5-5b shows the value prediction speedups for these three pipeline depths. The longer the pipeline, the higher the misprediction penalty. Therefore, with read stages ranging from 0 to 2, speedups vary from 16.4% to 12.2% with four clusters, from 8.5% to 5.9% with two clusters, and from 4.3% to 2.4% in a centralized architecture. We can conclude that processors with fewer stages between issue and exec will benefit more from this technique than other more deeply pipelined ones. More generally, this result shows that value prediction is quite sensitive to the misprediction penalty.

5.4.3 Value Predictor Table Size

The predictor table size determines the prediction accuracy, which has a significant influence on the performance. We have evaluated the impact of the predictor table size on a clustered architecture. Figure 5-7a shows the predictor rate and accuracy for several table sizes. With a 64K entry table, the prediction rate (number of predictions used by speculative instructions over

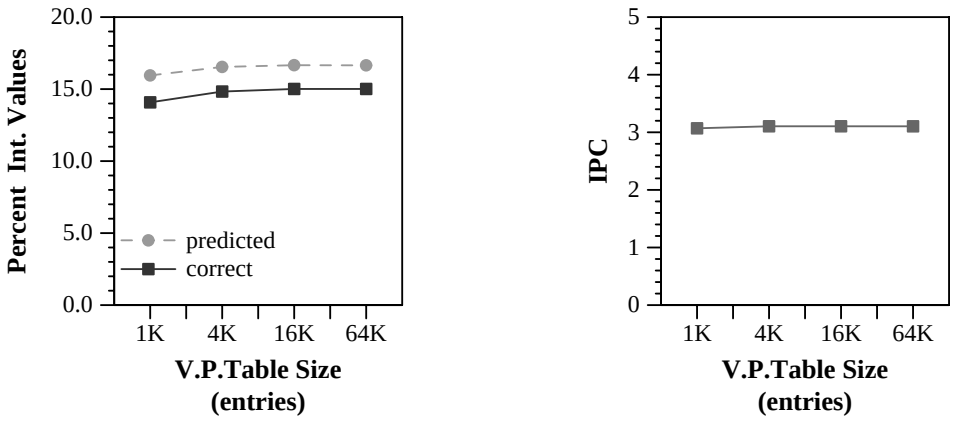


Figure 5-7: Impact of value predictor table size for 4 clusters (a) prediction rate and accuracy (b) IPC.

total number of integer values) is 16.6%, which is very low, due to predicting only load results and remote operands. The prediction accuracy is 90.1%. Reducing the table size to 1K entries reduces the prediction rate to 15.9% and the accuracy to 88.2%.

Figure 5-7b shows that on average, for a four-cluster configuration, there is less than 1.1% IPC degradation when the predictor table size is reduced from 64K to just 1K entries.

5.4.4 Experiments with the SpecInt95

In previous sections, the Mediabench [56, 62] benchmark suite was used for all the experiments. For the sake of higher generality of our conclusions, we run an identical set of experiments with the SpecInt95 [97] benchmark suite.

We found that the speed-ups of value prediction, with PRMB steering, are 6.7% for 4 clusters, 3.9% for 2 clusters and 3.6% for a conventional centralized architecture. We also found that, by using the enhanced VPB steering scheme, the speed-ups of value prediction are 7.5% for 4 clusters and 6.2% for 2 clusters.

Compared to the results with the Mediabench suite shown in previous sections, these results show similar trends, although the speed-ups are smaller. However, the same overall conclusions hold for both benchmark suites.

5.5 Conclusions

Future microprocessors are likely to be communication bound due to the increasing penalty of wire delays. In this chapter we showed that value prediction can be an effective instrument to improve communication locality. In particular, we have presented an approach to reduce inter-

cluster communication by means of a dynamic steering logic that leverages value prediction. Values produced in one cluster and consumed in another one may not require long wire delays to propagate from the producer to the consumer if the consumer can correctly predict the value. The validation required by the prediction is locally performed in the producer cluster.

We have shown that value prediction removes communications even for previously proposed steering schemes not specially designed to exploit value prediction. However, performance is higher if the steering logic exploits the predictability of values to improve the workload balance. We have presented a novel steering scheme (VPB) that avoids data dependence constraints on the assignment algorithm when a value is going to be predicted and there is a potential for improving the workload balance. Value prediction, together with VPB steering, removes on average 50% of the communications, and reduces substantially the workload imbalance, which translates into an average 14% IPC speedup in a four-clustered architecture. In contrast, an identical value predictor achieves only a 3% speedup in a centralized architecture.

We have also shown that this technique may produce even better improvements in future technologies, as wire delay - and hence communication latency - increases. The performance improvement of value prediction is quite sensitive to misprediction penalty, but it is less sensitive to the predictor table size, for the considered set of benchmarks and table sizes.

CHAPTER 6

EFFICIENT INTERCONNECTS FOR CLUSTERED MICROARCHITECTURES

In this chapter, we investigate the design of on-chip interconnection networks for clustered microarchitectures. This new class of interconnects have demands and characteristics different to traditional multiprocessor networks. In a clustered microarchitecture, a low inter-cluster communication latency is essential for high performance.

We propose some point-to-point interconnects and an improved instruction steering scheme, and show that they achieve much better performance than bus-based interconnects. The results show that the connectivity of the network together with effective steering schemes are key for high performance. We also show that these interconnects can be built with simple hardware and achieve a performance close to that of an idealized contention-free model.

6.1 Introduction

Previous work showed that the performance of a clustered superscalar architecture is highly sensitive to the latency of the inter-cluster communication network [16, 73]. Many steering heuristics have been studied to reduce the required communications [10, 16, 17, 70], and value prediction has been proposed to hide the communication latency [73]. The alternative approach proposed in this chapter consists of reducing the communication latency, by designing networks that reduce the contention delays and proposing effective improvements to the instruction steering scheme that minimize both the communication rate and the communication distance. Moreover, the proposed interconnects also reduce capacitance, thus speeding up signal propagation.

For a 2-cluster architecture it may be feasible to implement an efficient and contention-free cluster interconnect by directly connecting each functional unit output to a register file write port in the other cluster. However, as the number of clusters increases, the completely connected network may be very costly or unfeasible due to its complexity. On the other hand, a simple shared bus requires lower complexity but it has high contention. Therefore, a particular design needs to trade complexity for latency to find the optimal configuration.

Previous works on clustered microarchitectures have assumed interconnection networks that are either an idealized model ignoring complexity issues [10, 73], or they consider only 2 clusters (Multicluster [29], Alpha 21264 [53]), or they assume a simple but long-latency ring [3, 4, 52]. In this chapter, we explore several alternative interconnection networks with the goal of minimizing latency while keeping the cluster complexity low. To the best of our knowledge no other work has addressed this issue on dynamically scheduled processors. Sankaralingam et al. [86] analyzes several point to point interconnects for VLIW and Grid Processor architectures in a recent work, after our proposal [75]. We have studied two different technology scenarios: one with four 2-way issue clusters, the other with eight 2-way issue clusters. In both cases, we propose different point-to-point network topologies that can be implemented with low complexity and achieve performance close to those of idealized models without contention.

The rest of this chapter is organized as follows. In section 6.2 two new improvements to the steering scheme are proposed, section 6.3 discusses several design issues regarding the interconnection network, sections 6.4 and 6.5 describe the interconnect models proposed for four and eight clusters respectively, and section 6.6 analyzes the experimental results. Finally, section 6.7 summarizes the main conclusions of this chapter.

6.2 Improved Steering Schemes

The clustered architecture presented in this chapter assumes the best steering heuristic presented in section 4.2.7, i.e. the Priority RMB scheme with Accurate Rebalancing (AR-PRMB). As it was shown, the AR technique reduces communications during the periods when the workload imbalance exceeds the threshold, because it does not completely disable the communication criterion, but it just excludes the overloaded clusters from the choice of candidates. Since in this chapter we are going to explore architectures with up to 8 clusters, and communication latencies as high as 4 cycles, we expect that this technique proves even more effective than shown before. In more detail, this scheme works as follows:

1. To minimize communication penalties:
   1.1. If there is any unavailable operand, choose its producer cluster
   1.2. Else, select clusters with highest number of source registers mapped
2. Choose the least loaded among the above selected clusters

Except: If imbalance > threshold, exclude clusters with workload ≥ 0, prior to applying rules 1 and 2

6.2.1 A Topology-Aware Steering

For many of the interconnect topologies we study in this chapter, the latency of the communications depends on the distance between source and destination clusters. A topology-aware (TA) steering heuristic can take advantage of this knowledge to minimize the distance - and thus the latency - of the communications. Therefore, we have refined the primary criterion of the AR-PRMB algorithm to take the distance into account, in such a way that when all source operands are available (rule 1.2), it chooses the clusters that minimize the longest communication distance (the one that is in the critical path). To illustrate this feature, let us suppose that an instruction has two source operands, which are both available, and the left one is mapped to cluster 1, while the right one is mapped to clusters 2 and 3. In this case, the original primary criterion would select clusters 1, 2 and 3, since all of them have one operand mapped. Whatever is chosen, one copy would be needed, either between clusters 1 and 2 or between clusters 1 and 3. If we assume that cluster 1 is closer to cluster 2 than to cluster 3, then the topology-aware heuristic will consider only clusters 1 and 2.

6.3 The Interconnection Network

In this section we discuss several design trade-offs and constraints regarding the interconnection network, prior to describing in detail, in the following two sections, the schemes that have been experimentally analyzed for architectures with four and eight clusters.

6.3.1 Routing Algorithms

Interconnection networks have been widely studied in the literature for different computer areas such as multicomputers and network of workstations (NOWs) [23]. In these contexts, communication latencies may be thousands of processor cycles long, and routing decisions take several cycles. In contrast, for clustered microarchitectures performance is highly sensitive to the communication latency and just one cycle is a precious time, as shown by the results in section 6.6, and also by other previous works [16, 73]. Thus, in this context, networks must use simple routing schemes that carefully minimize communication latency (instead of maximizing throughput, like in other contexts). We assume that all routing decisions are locally taken at issue time (source routing), by choosing the shortest path to the destination cluster. If there is more than one minimal route, the issue logic chooses the first one that it finds available.

6.3.2 Register File Write Ports

Each cluster can inject copies to the network, which is connected to the cluster register files through a number of dedicated write ports where copies are delivered. From the point of view of the network design, including as many ports as required by its peak delivery bandwidth is the most straightforward alternative, but the number of write ports has a high impact on cluster complexity. First, each additional write port requires an additional result tag to be broadcast to the instruction issue queue, and the wakeup delay increases by a quadratic factor with respect

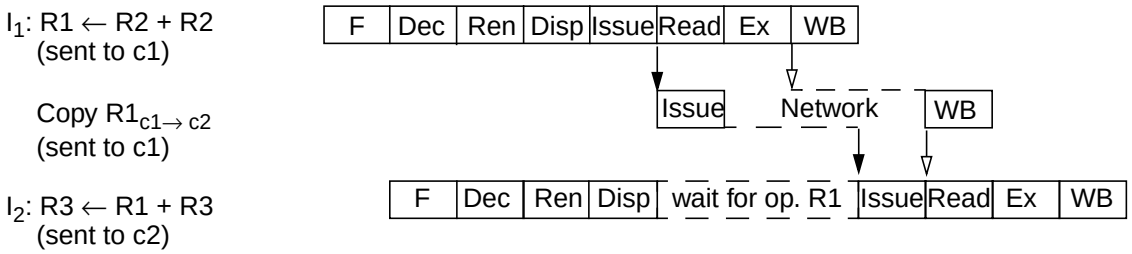


Figure 6-1: Sample timing of a communication between two dependent instructions I_1 and I_2, steered to clusters c1 and c2 respectively (solid arrows mean wakeup signals, hollow arrows mean data signals, and transmission time is 2 cycles in both cases).

to the number of broadcast tags [70]. Second, the register file access time increases linearly with the number of ports. Third, the register file area grows quadratically with the number of ports, which in turn makes the length and delay of the bypass wires to increase.

Moreover, previous studies (as well as our results in chapter 4) showed that, with adequate steering heuristics, the required average communication bandwidth is quite low (0.20 communications per instruction for 4 clusters [73]), and thus it is unlikely that having more than one write port per cluster connected to the network can significantly improve performance. Therefore, for all the analyzed networks we assume that they are connected to a single write port per cluster, except for the idealized models.

6.3.3 Communication Timing

In our distributed register file architecture, the access to remote operands is done exclusively through copy instructions, which are inserted into the instruction queues as normal instructions (see more details in chapter 2). A copy is issued when its source register is ready and it secures a slot of the network. Then, it reads the operand either from the register file or from the bypass, sends the value through the interconnection network, and delivers it to the consumer's cluster bypass network and register file.

The copy also sends through the network, along with the value, the tag of the destination physical register, in order to wake-up the dependent instructions. We assumed for simplicity that the tag forwarding delay is the same as the data forwarding delay. Consequently, the tag forwarding stays in the critical path of execution of the dependent instruction, which also includes issuing the copy instruction (see figure 6-1). Therefore, the total minimum issue distance between the producer and the consumer instructions equals the communication latency plus one cycle. However, a particular VLSI implementation could attempt to reduce this issue distance by optimizing the tag forwarding paths, which would leave it equal to the data communication latency.

6.3.4 Transmission Time

The total latency of a communication has two main components: the contention delays caused by a limited bandwidth, and the transmission time caused by wire delays. For a given network design, the first component varies subject to unpredictable hazards, and we evaluate it through simulation. On the other hand, the second component is a fixed parameter that depends on the propagation speed and length of the interconnection wires, which are low-level circuit design parameters bound to each specific circuit technology and design. To help narrowing this complex design space, we have taken two reasonable assumptions for point-to-point networks.

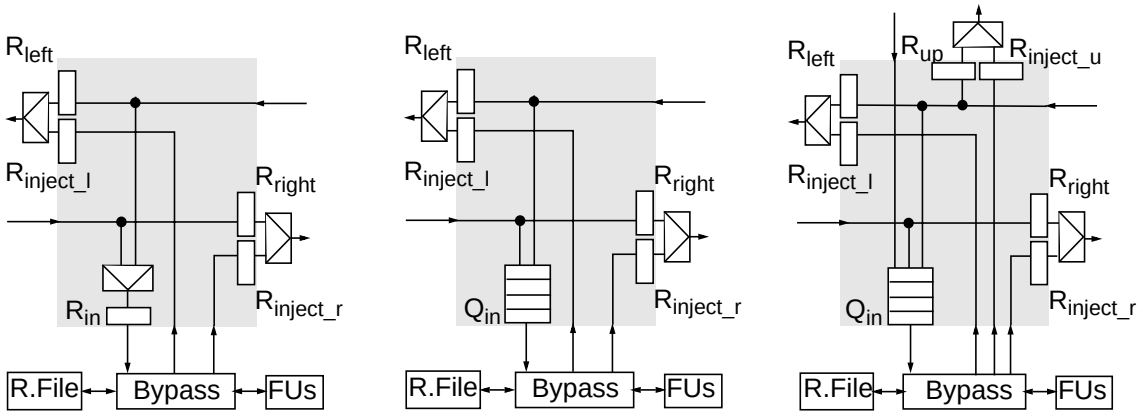
First, the minimum inter-cluster communication latency is one cycle. This clock cycle includes wire delay and switch logic delay. Note that, with current technology, most of the communication latency is wire delay. Second, only neighbor clusters (those at one-cycle distance) are directly connected with a pair of links, one in each direction. As a consequence, the communication between two non-neighbor clusters takes as many cycles as the number of links it crosses.

With these two assumptions, the space defined by different propagation speeds and wire lengths is discretized and reduces to the one defined by a single variable: the number of clusters that are at one-cycle distance from a given cluster (which is an upper bound of the connectivity degree of the network). Our analysis covers a small range of this design space by considering the connectivity degrees of several typical regular topologies.

Consistent with these long wire delays, the centralized L1 data cache is assumed to have a 3-cycle pipelined hit latency (address to cache, cache access and data back).

6.3.5 Router Structures

We assume a very simple router attached to each cluster for point-to-point interconnects. The router enables communication pipelining by implementing stage registers (buffers) in each output link (R_{right}, R_{left} and R_{up}, in figure 6-2). To reduce the complexity, the router does not include any other buffering storage for in-transit messages, but it rather guarantees that after receiving an in-transit message, it will be forwarded in the next cycle. This requirement is fulfilled by giving priority to in-transit messages over newly injected ones, and by structurally preventing that two in-transit messages compete for the same output link. Such a competence between in-transit messages never occurs on nodes with two neighbors, like those in a ring (figures 6-2a and 6-2b), but may happen in a mesh or a torus, where each node may have up to 4 neighbors (note, however, that since we have considered only small meshes with 4 and 8 clusters, each node has never more than 3 neighbors). For nodes with three neighbors, the router constrains the connectivity of in-transit messages by connecting every stage register (output link) to a single input link (see figure 6-2c), in the following way: in-transit messages can only traverse the router from the left to the right link, from the right to the left link or from the right to the upper link. Note that messages arriving from the upper link have no other connection than the input queue, thus this link is only available for messages doing their last hop.



(a) Connected to 2 nodes (sync. ring).

(b) Connected to 2 nodes (async. ring).

(c) Connected to 3 nodes (async. mesh and torus).

Figure 6-2: Router schemes for synchronous and asynchronous point-to-point interconnects

A *copy* instruction is kept in the issue queue until both its source operand is available and it secures the required injection register (R_{inject}, in figure 6-2), so no other buffering storage is required. That is, the scheduler handles the router injection registers as any other resource. However, while access requests for a bus-based network are sent to a distant centralized arbiter, the arbitration of each link in a point-to-point network is done locally at the source cluster, by simply choosing between one injection register and one stage register (priority is given to the latter one, as mentioned above). Eventually, the *copy* is issued and the outgoing message stays in one of the R_{inject} output registers while it is being transmitted.

The router also interfaces with the cluster datapath. For partially asynchronous networks, the router includes an input FIFO buffer (Q_{in}, in figures 6-2b and 6-2c) where all incoming messages are queued. Each cycle, only the message at the queue head is delivered to the cluster datapath, the others stay in the queue. For synchronous networks, the router is still less complex. By appropriately scheduling the injection of messages at the source cluster (more details are given later), the proposed scheme guarantees that a given router does not receive more than one input message per cycle. Therefore, the router requires just a single register (R_{in}, in figure 6-2a), instead of the FIFO buffer.

6.3.6 Bus versus Point-to-Point Interconnects

Although our analysis mainly focuses on point-to-point networks, we also study a bus interconnect, for comparison purposes. It is made up of as many buses as clusters, each bus being connected to a write port in one cluster, and each cluster being able to send data to any bus (figure 6-3a). Although this is a conceptually simple model, it has several drawbacks that make it little scalable. First, since buses are shared among all clusters, their access must be arbitrated, which makes the communication latency longer, although bandwidth is not affected as long as arbitration and transmission use different physical wires. Second, a large portion of

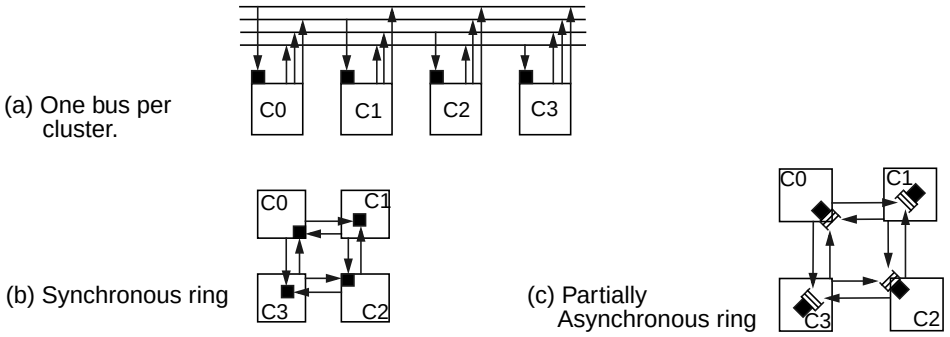


Figure 6-3: Four-cluster topologies

the total available bandwidth, which is proportional to the number of clusters, is wasted due to the low bandwidth requirements of the system. However, if the number of buses was reduced, then the number of conflicts would increase, and hence the communication latency. Third, each bus must reach all clusters, which implies long wires and long transmission times, which can drastically reduce the bandwidth if the bus transmission time is not pipelined[1].

Compared to the above bus interconnect, a point-to-point interconnect (a ring, a mesh, a torus, etc.) has the following advantages. First, the access to a link can be arbitrated locally at each cluster. Second, communications can be more easily and effectively pipelined. Third, delays are shorter, due to shorter wires and smaller parasitic capacitance (there are less devices attached to a point-to-point link than to a bus). Fourth, network cost is lower than a configuration with as many buses as clusters. Finally, it is more scalable: when the number of clusters increases, its cost, bandwidth and ease of routing scales better than for the bus-based configuration.

6.4 Four-Cluster Network Topologies

For four clusters, we propose two alternative point-to-point networks based on a ring topology, and compare them to a realistic bus-based network (see figure 6-3). We also compare their performance to that of an idealized ring, which represents an upper bound for ring networks. Below we describe these topologies.

6.4.1 Bus2

This is a realistic bus interconnect with a 2-cycle transmission time (hence its name). It has as many buses as clusters, each one connected to a single write port (see figure 6-3a), and a very simple centralized bus arbiter. The total communication latency is 4 cycles because bus

1. Note that it is difficult to pipeline bus communications, but it is easy to pipeline communication through point-to-point links (although the latter case is not needed with current VLSI technology, so we assume a transmission time of 1 cycle per hop), it clearly indicates that point-to-point links are much more scalable than buses.

arbitration, including the propagation of the request and grant signals, takes 2 additional cycles. We assume that the arbitration time may overlap with the transmission time of a previously arbitrated communication, so each single bus bandwidth is 0.5 communications per cycle.

6.4.2 Synchronous Ring

This interconnect is one of the contributions of this work, since previously proposed rings work in asynchronous mode. This topology assumes no queues in the routers, neither to store in-transit messages nor to store messages arriving at their destination clusters.

Since no queues are included at the destination clusters, when a message arrives it must be immediately written into the register file. The router arbitration logic injects copy instructions with an algorithm (summarized in table 6-1) that ensures that no more than one message arrives at a time at a given node. During odd cycles, a source cluster *src* is allowed to send a short-distance message (D=1) to its adjacent cluster in the clockwise direction (*(src +1) mod 4*), and a long-distance message (D=2) in the counter-clockwise direction (*(src + 2) mod 4*). During an even cycle, the allowed directions are reversed. Since in-transit messages are given priority over

Table 6-1: Rules to secure a link in the source cluster *src* (D refers to distance in cycles)

Direction	Odd Cycle		Even Cycle	
	D	Target Cluster	D	Target Cluster
Clockwise	1	*src → (src+1) mod 4*	2	*src → (src+2) mod 4*
Counter-clockwise	2	*src → (src+2) mod 4*	1	*src → (src+3) mod 4*

newly injected ones (see section 6.3.5), an issued copy instruction may have to wait in the injection register until the cycle parity is appropriate.

Despite the fact that there are cyclic dependencies between links [23], deadlocks are avoided by synchronously transmitting messages through all the links in the ring, even if the stage buffer at the next router is busy (it will be free when the message arrives). This is possible thanks to using the same clock signal for all the routers and giving a higher priority to in-transit messages.

6.4.3 Partially Asynchronous Ring

Typical asynchronous networks include buffers both in the intermediate routers, to store in-transit messages, and in the destination routers, to store messages that are waiting for a write port (in our case to the register file) [23]. The former are removed in our design, like in the synchronous ring. However, we still need the latter, since two messages can arrive at the same time to the same destination cluster and there is only one write port in each cluster. In this case, the message whose data cannot be written is delayed until it has a port available. Note that the system must implement an end-to-end flow control mechanism in order not to lose messages when a queue is full. This is an additional cost of the asynchronous schemes, which is discussed in more detail in section 6.6.3. In this network, routers use the same clock signal. Therefore, it

is only partially asynchronous. A fully asynchronous network has not been considered because its cost would be much higher (larger buffers, link-level flow control, extra buffers to avoid deadlocks, etc.). Deadlocks are avoided as in the synchronous ring.

6.4.4 Ideal Ring

For comparison purposes, we consider an idealized ring whose inter-cluster distances are the same as those of the realistic ring (as discussed previously), but an unlimited bandwidth is assumed, which makes it contention-free (i.e., it has an unlimited number of links between each pair of nodes and an unbounded number of register file write ports for incoming messages in each cluster). The performance of this ideal ring lets us estimate how much performance is lost due to interconnect bandwidth constraints.

6.4.5 Ideal Crossbar

We consider also an idealized crossbar with unlimited bandwidth and 1-cycle distance between any pair of clusters. Its performance is an upper bound for all other models, and it lets us estimate how much performance is lost due to constraining the connectivity degree - and hence the complexity - to 2 adjacent nodes per cluster.

6.5 Eight-Cluster Network Topologies

For eight-cluster architectures, we first consider two ring-based interconnects, synchronous and partially asynchronous, similar to those proposed for 4-cluster architectures, and also two versions of a realistic bus-based network, having transmission times of 2 and 4 cycles, respectively.

In addition to the ring, we also analyze mesh and torus topologies, both of them partially asynchronous, since they feature lower average communication distances. Figure 6-4 shows these two new schemes. Below, we describe each scheme in detail.

6.5.1 Bus2 and Bus4

The bus required to connect 8 clusters is likely to be slower than that required by the 4-cluster configuration due to longer wires and higher capacitance. To account for this, we consider two bus-based configurations: the Bus2, which optimistically assumes the same latencies as those of the 4-cluster configuration (i.e., a transmission time of 2 cycles), and the Bus4, which more realistically assumes twice this latency (i.e., a transmission time of 4 cycles).

6.5.2 Synchronous and Partially Asynchronous Rings

For this interconnect, the scheduling algorithm of the synchronous ring discussed in section 6.4.2 for 4 clusters is extrapolated to 8 clusters. Like in the 4-cluster configuration, at issue time the scheduler of copy instructions (shown in table 6-2) must ensure that only one message arrives at a time at a given cluster, because there is only one write port to the register file. Note that distances in an 8-cluster ring topology range from 1 (D=1) to 4 (D=4) cycles. The partially asynchronous ring model is identical to that described for four clusters in section 6.4.3.

Table 6-2: Rules to secure a link in the source cluster *src* (D refers to distance in cycles)

Direction	Odd Cycle		Even Cycle	
	D	**Target Cluster**	**D**	**Target Cluster**
Clockwise	1	*src → (src+1) mod 8*	2	*src → (src+2) mod 8*
	3	*src → (src+3) mod 8*	4	*src → (src+4) mod 8*
Counter-clockwise	4	*src → (src+4) mod 8*	3	*src → (src+5) mod 8*
	2	*src → (src+6) mod 8*	1	*src → (src+7) mod 8*

6.5.3 Mesh

A mesh topology (see figure 6-4a) reduces some distances with respect to a ring. The average distance in a ring is 2.29 hops, while in a mesh it is 2 hops; however, the maximum distance is still 4 hops. The dashed lines in the figure show the links added to the ring topology to convert it into a mesh.

Due to the increased connectivity, this topology introduces a new problem to the design of the routers with respect to a ring, because at central nodes (labelled C2, C3, C6 and C7) more than one in-transit message at the router input links could compete to access the same output link. As discussed in section 6.3.5, our approach is to constrain the connectivity of in-transit messages within the router. On the one hand, an *upper* router input (refer to figure 6-2c) is only connected to the input queue, so the routing algorithm must ensure that this link is used only for the last hop of a transmission. The four links between C2-C3 and C6-C7 in our mesh (shown with dashed arrows in figure 6-4a) are connected to *upper* router inputs and outputs. Thus, if one message is sent, for instance, from cluster C2 to C7, it must be routed through C6 because the link C2-C3 is not associated to the link C3-C7. On the other hand, in-transit messages arriving to a *right* router input have no constraints, so they may be routed either to the *left* or to the *upper* output. The four links between C3-C7 and C2-C6 in our mesh (figure 6-4a) are connected to *right* router inputs and outputs. Finally, the rest of links connected to the four central nodes of the mesh are connected to *left* router inputs. In-transit messages arriving to a *left* router input can only be routed to the *right* output. Thus, for instance, a message from C0 to C3 must be routed through C1 because the link C0-C2 is not associated to the link C2-C3.

Again, deadlocks are avoided by using the same clock signal for all the routers and transmitting messages synchronously.

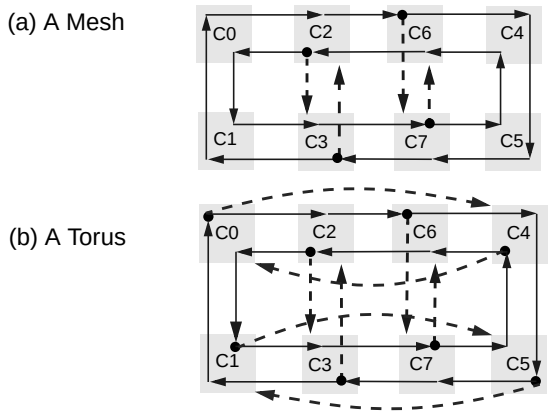


Figure 6-4: Additional topologies for 8 clusters. A black dot at the end of a link means that there is more than one link that can be followed for the next hop if the corresponding node is not the destination. Messages can always be routed through solid links but dashed links are only used for their last hop.

6.5.4 Torus

A torus has smaller average distance than a mesh (see figure 6-4b). In all nodes, more than one in-transit message at the router input links could compete to access the same output link. Like for the mesh, this problem is solved without including intermediate buffers, by constraining the connectivity of several links. The solution is outlined in the figure, where dashed arcs indicate links with a limited connectivity (see also router details in figure 6-2c).

Note that this constraint does not change the minimal distance between every pair of nodes, but for some pairs it does reduce the number of alternative routes. For example, there is only one 2-hop route from C4 to C1. However, due to the poor utilization of the network (as we will show later) this is a minor drawback.

Again, deadlocks are avoided as indicated above. On another issue, when mapping torus links on silicon, some links may be longer than the rest (e.g., links between C0 and C4). This may introduce delays in those particular links. For the sake of simplicity, we did not consider that additional delay.

6.5.5 Ideal Torus

For comparison purposes, we also consider an idealized torus model, with distances identical to those of the realistic torus but with unlimited bandwidth, which makes the network to be contention-free. In other words, it is assumed that the network has an unlimited number of links between any pair of adjacent nodes and an unbounded number of register file write ports connected to the network in each cluster. The performance of this model is an upper bound on the performance of the realistic torus.

6.5.6 Ideal Crossbar

We consider also an idealized crossbar with unlimited bandwidth and 1-cycle distance between any pair of clusters. Its performance is an upper bound for all other models, and it lets us estimate how much performance is lost due to constraining the connectivity degree.

6.6 Experimental Evaluation

In this section the different network architectures proposed above are evaluated. For these experiments, we have used the Mediabench benchmark suite [56, 62] and assume all the experimental setup described in section 2.3, including the simulator and the architectural parameters in table 2-1. The eight-clusters architecture assumed an identical cluster model as those of the four-clusters architecture, which means doubling the total effective issue width of the processor. Accordingly, the eight-cluster architecture also assumes twice the fetch/decode bandwidth, number of data cache ports and number of entries in the reorder buffer and in the load/store queue.

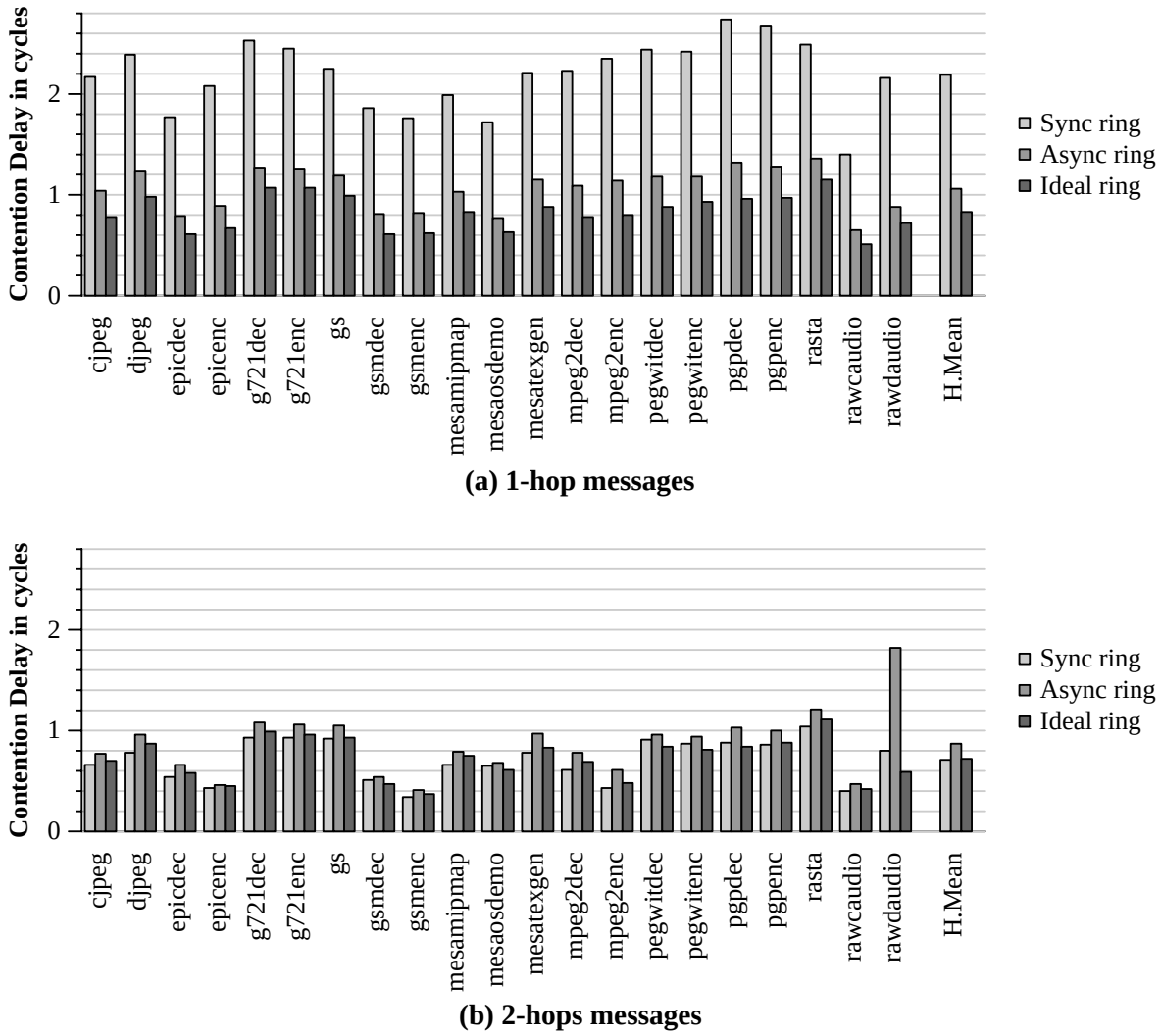
6.6.1 Network Latency Analysis

To gain some insight on the different behavior of synchronous and partially asynchronous rings for a 4-cluster architecture, we analyze their average communication latency. In particular, since the transmission time component of the latency is the same for both interconnects, we only analyze the contention delay component.

Figure 6-5 compares the communications contention delay for each of the two ring interconnects, and it also includes an ideal ring for comparison. For each of the former two interconnects, the contention delay has two components: it may be caused by an insufficient issue width or an insufficient interconnect bandwidth. In contrast, the contention delay of the ideal ring is exclusively due to the limited issue width, and it is on average 0.8 cycles. Therefore, comparing the delays of the first two interconnects to that of the ideal-ring, the difference gives an estimation of the contention caused by the insufficient interconnect bandwidth.

As shown in figure 6-5, short-distance messages (graph a) wait for longer than long-distance ones (graph b). The main reason is the available bandwidth for each type of message: the latter have two alternative minimal-distance routes, while the former have only one. However, since the routing algorithm is the same for both ring interconnects, it does not explain the differences observed between the two rings.

Figure 6-5a shows that the contention delay of short-distance messages for a synchronous ring (2.19 cycles) is two times longer than for a partially asynchronous one (1.06 cycles). In contrast, the contention caused to long-distance messages for a synchronous ring (0.71 cycles) is just slightly lower than for a partially asynchronous one (0.87 cycles). These differences are



(a) 1-hop messages

(b) 2-hops messages

Figure 6-5: Average contention delays of 1-hop and 2-hops messages, with synchronous, partially asynchronous and ideal ring interconnects

due to the different ways each interconnect avoids conflicts between messages that require access to the same register file write port: in a synchronous ring, these conflicts are prevented by ensuring that a short-distance message is not issued if the parity of the cycle is not the appropriate one (see table 6-1), regardless of whether the link is busy or not. For example, a long-distance message from C1 to C3 (see figure 6-3b) will be injected in an even cycle, thus reaching the router at C2 and requesting the C2-C3 link during the next odd cycle. As messages from C2 to C3 must be injected during odd cycles, these short-distance messages will be delayed if there are in-transit long-distance messages. In a partially asynchronous ring, a message of any kind can be issued as soon as the required output link is available, although it may have to wait in the destination cluster router until it gains access to the register file write port. For an asynchronous ring, the network contention causes a delay between 0.15 and 0.23 cycles, while the rest of the contention delay of the communications is due to the issue width.

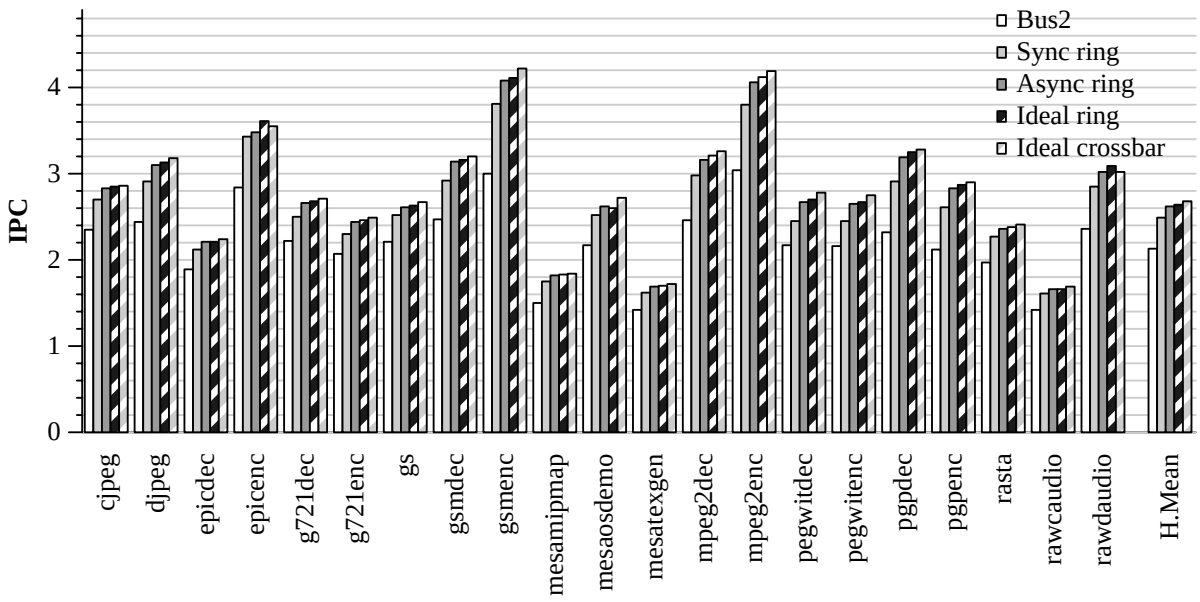


Figure 6-6: Comparing the IPC of 4-cluster interconnects

To summarize, the long delays caused to short-distance messages by the scheduling constraints of the synchronous ring make its overall contention delay be higher than for a partially asynchronous one. In addition, since there are about twice as many short-distance messages than long-distance ones, they have a high impact on the overall contention delay. As a consequence, the partially asynchronous ring performs better than the synchronous one, as it is shown below.

6.6.2 Performance of Four-Cluster Interconnects

Figure 6-6 compares the performance, reported as number of committed instructions per cycle (IPC), of a four-cluster architecture for all the proposed interconnects.

The two ring interconnects consistently achieve better performance than the bus topology, for all benchmarks. This is mainly because short-distance messages have a shorter transmission time in point-to-point interconnects, and because the steering heuristic exploits it to keep close instructions that have to communicate. Besides, the ring topology offers a higher bandwidth, although in this scenario bandwidth is not critical for performance due to the low traffic generated by the steering scheme [73] (e.g. it is on average 0.20 communications per instruction, with a partially asynchronous ring).

The IPC achieved by the synchronous ring is, on average, 16.8% higher than that achieved by the bus, while the partially asynchronous ring performance is 23.2% higher than that of the bus, because the contention delays of short-distance messages are lower for the asynchronous ring, as discussed in section 6.6.1.

The performance of the partially asynchronous ring is very close to that of the ideal ring (less than 1% difference), which shows that increasing the number of links or the number of

register file write ports would hardly improve performance. In other words, due to the effectiveness of the steering logic to keep the traffic low, a simple configuration with two links between adjacent clusters (one in each direction) and a single register file write port for incoming messages is clearly the most cost-effective design.

Finally, we found that a ring performs very close to a complex fully connected crossbar. The performance lost by reducing the connectivity degree from 3 to 2 adjacent nodes per cluster (corresponding to the ideal crossbar and the ideal ring, respectively) is on average just 1.3%. In other words, a ring interconnect is a cost-effective topology for a four cluster interconnect, since having a higher connectivity - hence complexity - returns very small performance gains.

6.6.3 Queue Length

Typical partially asynchronous rings need specific mechanisms to prevent (or to recover from) potential overflows of the network buffers. In our partially asynchronous interconnect, this problem occurs only in the queues for incoming messages at each cluster.

In order to adequately dimension these queues, we first assumed unbounded size queues and measured the number of occupied entries each time a new message arrives at its destination cluster. Note that with FIFO queues and a single write port, this number is equal to the number of cycles a message stays in the queue. We found that for any benchmark, more than 85% of the messages do not have to wait because they find the queue empty (92.1%, on average), and the maximum observed number of occupied entries was 11. For instance, table 6-3 shows a typical queue length distribution (for benchmark *djpeg*).

Table 6-3: Queue length distribution (for *djpeg*)

# occupied entries	# messages	Distribution (% times)	Cumulative Distribution (%)
0	1263899	90.25	90.25
1	122802	8.77	99.02
2	12078	0.86	99.88
3	1524	0.11	99.99
4	105	0.01	100.00
5	18	0.00	100.00
6	4	0.00	100.00
>= 6	0	0.00	100.00

Although 11-entry queues are long enough in our experiments, the model should ensure that data is never lost, in order to guarantee execution correctness. Two approaches are possible: first, to implement a flow control protocol that prevents FIFO queue overflows; and second, to implement a recovery mechanism for these events. Flow control can be based on credits. In this case, each cluster would contain a credit counter for each destination cluster. Every time a message is transmitted to a cluster, the corresponding credit counter would be decreased. If the counter is equal to zero, the message would not be transmitted because the FIFO queue may be full. When a message is removed from the queue, a credit is returned to the sender of that message, thus, consuming link bandwidth. Upon reception of the credit, the corresponding

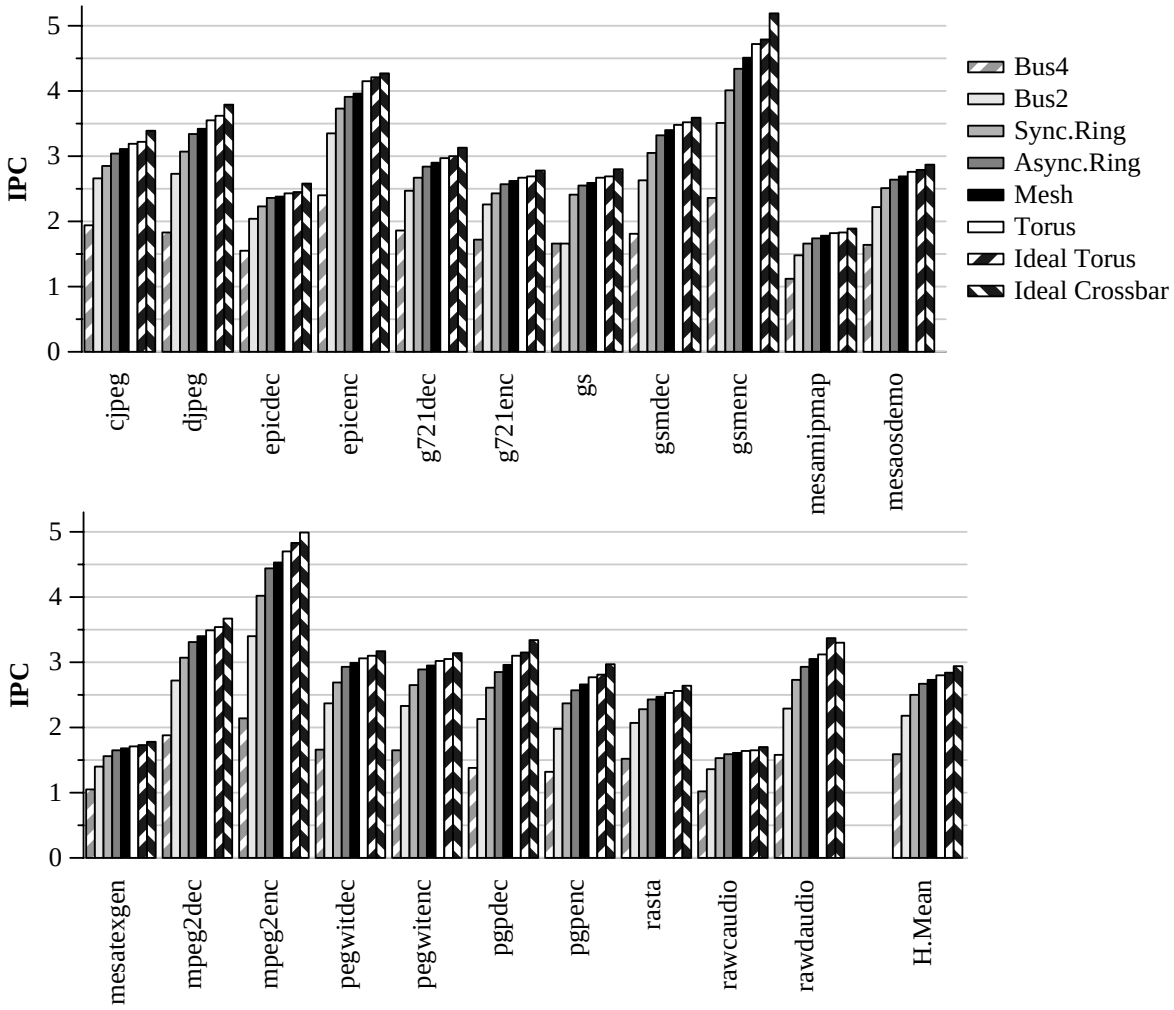


Figure 6-7: Comparing the IPC of 8-cluster interconnects

credit counter is increased. However, since overflows are so infrequent, the most cost-effective solution in case of an overflow is to squash the instruction that generated the message that caused the overflow, as well as all other younger instructions, very much like in the case of exceptions or branch mispredictions, and to restart again execution at this instruction. This approach requires minimal additional hardware and it produces negligible performance penalties (for 11-entry queues there is no penalty at all for our benchmarks).

6.6.4 Performance of Eight-Cluster Interconnects

In this section, we evaluate the eight-cluster network interconnects described in section 6.5, for a 16-way issue architecture (as described at the beginning of section 6.6). Figure 6-7 shows the IPC for the different schemes. The point-to-point ring achieves a significant speed-up even over the optimistic bus architecture denoted as bus2. The average speed-up of the synchronous ring over bus2 is 14.6% whereas the partially asynchronous ring outperforms bus2 by 22.3%.

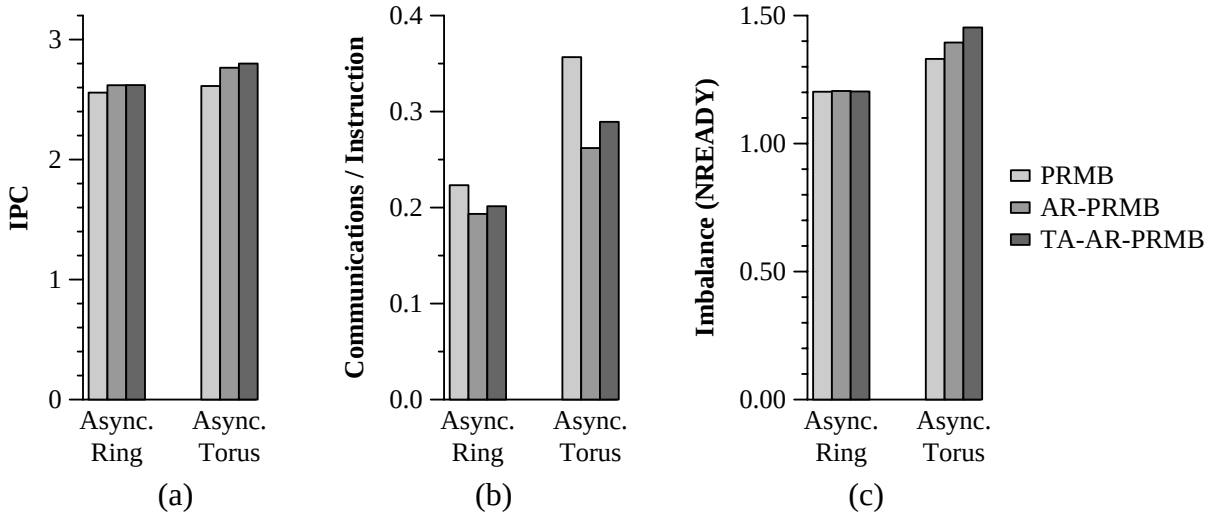


Figure 6-8: Effectiveness of the Accurate Rebalancing (AR) and Topology Aware (TA) techniques applied to the baseline PRMB steering scheme, for a four-cluster ring and an eight-cluster torus: (a) IPC, (b) Communications rate and (c) Workload imbalance

Comparing the partially asynchronous topologies, the mesh achieves an IPC 2.2% higher than that of the ring, while the IPC of the torus is 4.9% higher than that of the ring. On the other hand, the partially asynchronous torus performance is very close to that of the ideal torus configuration with unlimited bandwidth (just 1.4% difference). The performance of the ideal torus is just 3.4% below that of the ideal crossbar that has unlimited bandwidth and all nodes at a 1-cycle distance.

6.6.5 Effectiveness of the Accurate-Rebalancing and Topology-Aware Steering

In all the previous experiments it was assumed the PRMB steering scheme, including both the Accurate-Rebalancing (AR) and the Topology-Aware (TA) improvements described in section 6.2. In this section, the effectiveness of these two techniques are analyzed. They are evaluated for a four-cluster architecture with an asynchronous ring and for an eight-cluster architecture with a torus interconnect. Figure 6-8 compares the PRMB steering with and without these two techniques. It is shown the average IPC (a), the average communications rate (b), and the average workload imbalance (c).

The AR technique is aimed at reducing the communications generated during strong imbalance situations, when the PRMB steering is mainly concerned on rebalancing the workload. Instead of totally ignoring dependences, it just excludes the overloaded clusters. As shown in graph (a), AR improves the performance of PRMB by 2.4% and 5.8% with four and eight clusters respectively, because it significantly reduces the amount of communications. As shown in graph (b), AR reduces the communication rate by 13% and 27% with four and eight clusters respectively. Not surprisingly, the effectiveness of the AR technique grows with the number of clusters, because the likelihood of causing a communication when operand locality is ignored increases with the number of clusters.

The TA technique is aimed at minimizing communication distances - hence latencies - for point-to-point interconnects. As shown in figure 6-8a, it produces a small 1.2% IPC improvement over the AR-PRMB scheme for eight clusters and almost no effect for four clusters (the total improvement using both AR and TA techniques is 7.2% and 2.5% respectively). TA produces a small impact on performance because there are actually few instructions that offer the chance to reduce the communication distance, and because in some of these cases the distance is reduced at the expense of generating one extra communication.

With our AR-PRMB steering scheme, the opportunity to reduce the communication distance occurs only when an instruction has two register operands, and both are available, and they are mapped in two disjoint subsets of clusters. When this happens, at least one communication is required, and the TA technique chooses the cluster that minimizes the longest communication distance to the source operands instead of choosing one of the clusters with a source register mapped. We found that the TA reduces the average communication distance from 1.32 to 1.20 hops for four clusters, and from 1.70 to 1.37 hops for eight clusters. However, it often occurs that the TA chooses a cluster where none of the operands is mapped, which forces generating a second communication. We found that the TA increases the number of communications per instruction from 0.193 to 0.201 for four clusters, and from 0.262 to 0.289 for eight clusters, as shown in figure 6-8b. Therefore, the added communication overhead offsets the expected improvements of reducing communication latency.

6.6.6 Experiments with the SpecInt95

In previous sections, the Mediabench [56, 62] benchmark suite was used for all the experiments. For the sake of higher generality of our conclusions, we run an identical set of experiments with the SpecInt95 benchmark suite [97].

For 4 clusters, we found that the synchronous ring performs on average 12.6% better than the bus2, while the partially asynchronous ring outperforms bus2 by 14.9%. The performance of the partially asynchronous ring is very close to that of the ideal ring with unlimited bandwidth (1.0% difference), and it is just 1.7% below that of the ideal crossbar, with unlimited bandwidth and 1-cycle latency between any pair of nodes.

For 8 clusters, the average speed-up of the synchronous ring over bus2 is 9.2% whereas the partially asynchronous ring outperforms bus2 by 13.5%. Comparing the partially asynchronous interconnects, the mesh achieves an IPC 1.5% higher than that of the ring, while the IPC of the torus is 3.3% higher than that of the ring. On the other hand, the partially asynchronous torus performance is just 0.4% below that of the ideal torus configuration with unlimited bandwidth, and 4% below that of the ideal crossbar with unlimited bandwidth and 1-cycle latency between any pair of nodes.

Finally we found that the AR technique improves the performance of the PRMB steering scheme for an 8-cluster torus and a 4-cluster ring by 2.5% and 1.1% respectively. The TA technique produces a 1.4% speedup for 8 clusters and almost no impact for 4 clusters. The total performance improvement using both the AR and TA techniques for an 8-cluster torus is 4%.

Compared to the results with the Mediabench suite shown in previous sections, these results show similar trends, although in some cases the differences among the various configurations may vary. However, the same overall conclusions hold for both benchmark suites.

6.7 Summary and Conclusions of this Chapter

In this chapter we have investigated the design of on-chip interconnection networks for clustered microarchitectures. This new class of interconnects have demands and characteristics different to traditional multiprocessor networks, since a low communication latency is essential for high performance. We have shown that simple point-to-point interconnects together with effective steering schemes achieve much better performance than bus-based interconnects. Besides, the former do not require a centralized arbitration to access the transmission medium.

In particular, we have proposed a very simple synchronous ring interconnect that only requires five registers and three multiplexers per cluster and substantially improves the performance of a bus-based scheme.

We have also shown that a partially asynchronous ring performs better than the synchronous one at the expense of some additional cost/complexity due to the additional queue required per cluster. However, we have found that a tiny queue will practically never overflow. Thus, instead of using complex flow control protocols, it is much more cost-effective to handle overflows by flushing the processor pipeline, which is a mechanism that current microprocessors already implement for other purposes (e.g., branch misprediction).

We have explored other synchronous and partially asynchronous interconnects such as a mesh and a torus, in addition to rings. These three topologies basically differ in their connectivity degree, and consequently, in the average inter-cluster distances. From our study we extract two main conclusions. First, the interconnects with higher connectivity perform better because they have shorter communication latency. However, point-to-point partially asynchronous interconnects with moderate connectivity/complexity perform close to an idealized crossbar with unlimited bandwidth and all nodes at one-cycle distance: a four-cluster ring performs within 2% of the ideal, and an eight-cluster torus performs within 4% of the ideal. Second, despite the low hardware requirements of partially asynchronous interconnects, they achieve a performance close (within 1%) to an equivalent idealized interconnect with unlimited bandwidth and number of write ports to the register files.

To conclude, the choice of an effective interconnection network architecture together with an efficient steering scheme is a key to high performance in clustered microarchitectures. Simple implementations of point-to-point interconnects such as those proposed in this chapter are quite effective and scalable.

CHAPTER 7

A CLUSTERED FRONT-END FOR SUPERSCALAR PROCESSORS

This chapter studies techniques for distributing the main components of the processor front-end with the goals of reducing their complexity and avoiding replication, so they extend the advantages of clustering to structures like the branch prediction, instruction fetch, steering and renaming. In particular, effective techniques are proposed to cluster the branch predictor and the steering logic, which minimize the wire delay penalties caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation, and the cluster assignment. By reducing the latency of these critical loops, clustering results in faster clock rates, or bigger structures with the same clock rate. The schemes proposed in this paper to deal with inter-cluster communications are very effective since they reduce communication penalties to just a 4% IPC degradation, for SpecInt95.

7.1 Introduction

Clustering of computational elements [42, 70, 99, 105] is an effective method for dealing with scaling, complexity [70], power [112], heat, and clock distribution [47] problems. Previous clustered microarchitecture proposals [18, 29, 34, 52, 53, 54, 70, 73, 82, 112] have focused on clusters containing register files, functional units, and issue queues. However, in that previous work, the processor front-end units (e.g. instruction fetch, decode, rename) are centralized in a conventional manner. However, the advantages of clustering apply to any processor component. In the research reported here, we propose and study the design of front-ends containing clustered subsystems. The branch predictor, instruction cache, decode and rename logic are all either distributed, or in some cases replicated, and grouped into a number of clusters.

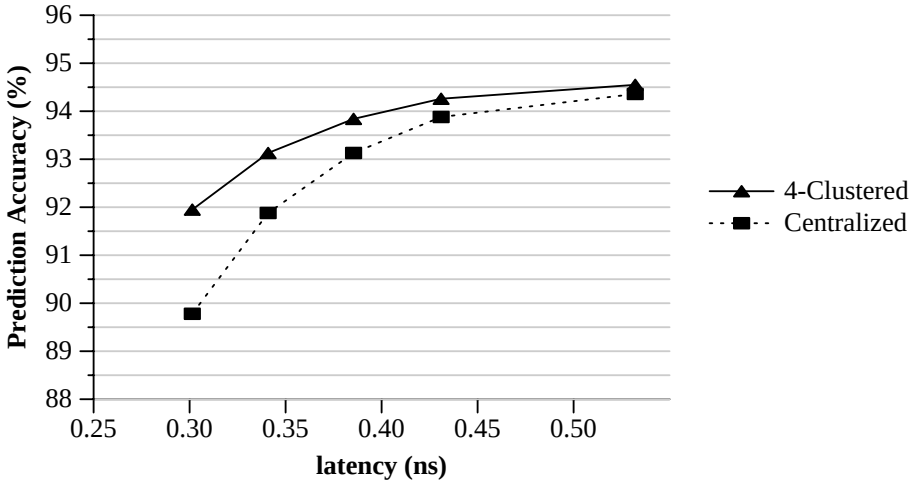


Figure 7-1: Accuracy vs. latency for a centralized and a 4-clustered predictors.

Note that the advantages of clustering apply to any processor component. For instance, let us consider the branch predictor. Because it is involved in the task of the fetch address generation, which forms a critical hardware loop, the predictor size -hence its accuracy- is greatly limited by cycle time. By partitioning the predictor into four smaller predictors, the latency of each one is considerably lower. Conversely, the effective size of a clustered branch predictor can be made four times bigger without increasing its latency. Figure 7-1 illustrates this by comparing the effectiveness of a clustered branch predictor to that of a centralized one, for different predictor latencies (sizes)[1]. As it can be seen, clustering results in important accuracy improvements for a given latency, or significant latency reductions for a given accuracy. There are many techniques, such as banking, that may optimize the access time of a predictor table, but they are orthogonal to our approach, because they equally apply to both a centralized predictor and each partition. Our approach to partitioning the branch predictor goes one step further because it leaves cross-structure wire delays out of the critical path of the fetch address generation loop by converting them to cross-cluster communications.

As a second example, let us consider the steering logic in a superscalar architecture with a clustered back-end. This piece of logic is located in the front-end and takes care of selecting the most appropriate execution cluster for each instruction, prior to steering it to the issue queues. A cluster assignment algorithm that steers instructions according to data dependences is key for performance, especially if the register file is also distributed. Unfortunately, this kind of algorithm serializes the steering of a sequence of instructions because the assignment of each instruction depends on previous assignments. For an eight-way or wider superscalar, this task may possibly take more than a single cycle. However, because the assignments of one fetched block of instructions are needed to steer the next block, the steering task cannot be spread into several stages without generating pipeline bubbles. Our approach is to partition the assignment

1. This experiment assumes a gshare+bimodal hybrid predictor with equally sized tables. Table sizes range between 0.25K and 64K entries. The latencies are calculated for a 90nm tech. and for layouts optimized for speed, using a modified version of the CACTI 3.1 tool (http://research.compaq.com/wrl/people/jouppi/CACTI.html).

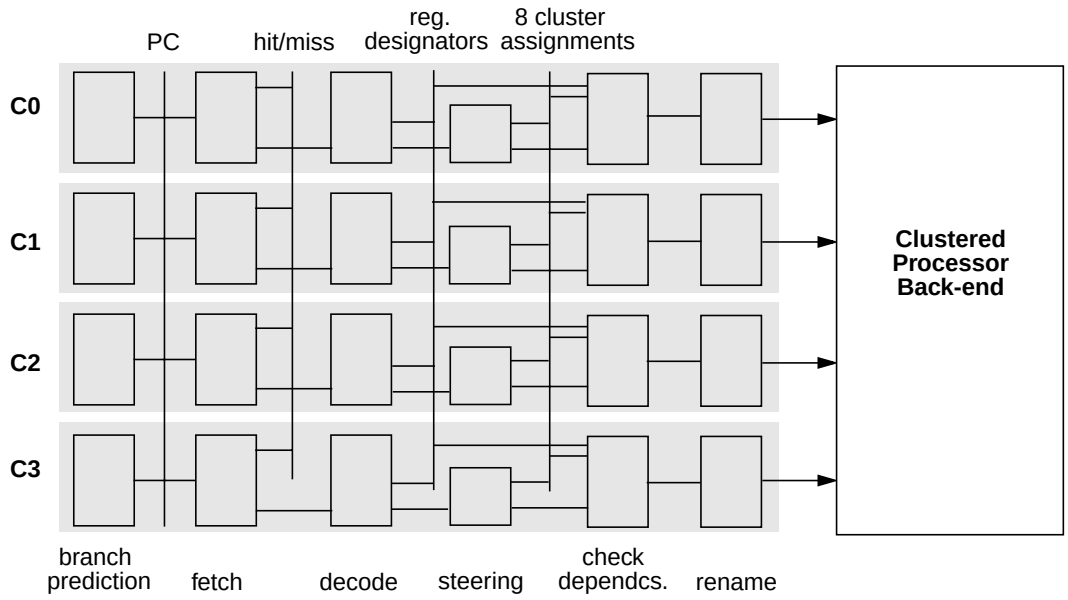


Figure 7-2: Partitioning an 8-way processor's front-end into four clusters.

task into smaller units that operate in parallel, thus reducing the amount of work of each partition and shortening the total latency of the cluster assignment logic.

To summarize, in this work we explore techniques for clustering each of the major front-end components with the objective of minimizing replication and inter-cluster communication. Even if a designer chooses not to implement a fully clustered front-end, this research points to new ways of partitioning the individual front-end components, such as the branch predictor, that could be used in an otherwise conventional front-end.

We evaluate the performance impact of clustering front-end structures by comparing with a baseline processor composed of a conventional centralized front-end and a clustered back-end (similar to the one described in chapter 2). To account for longer inter-cluster delays, we follow the strict rule of adding a full clock cycle to any paths that involve inter-cluster communication. Ignoring the clock cycle advantage, the proposed clustered front-end has about the same performance as the non-clustered one. For instance, we observe that the average IPC of the clustered organization for SpecInt95 is within 4% of the non-clustered one.

7.2 Clustering Front-End Subsystems

As a vehicle for developing and studying clustered front-end microarchitectures, we begin with a conventional eight-way superscalar microarchitecture consisting of a clustered back-end and a front-end divided into four clusters, each one capable of processing two instructions per cycle (see Figure 7-2). Consequently, all our discussion will focus on this particular 4-by-2 configuration but the mechanisms can be applied to any other (m-by-n) configuration.

Because the I-cache and branch predictor tables are *partitioned* among the clusters, each front-end cluster holds its own PC, a portion of the I-cache and a portion of the branch predictor. The rename table, in contrast, is *replicated* in all clusters, but each copy has fewer read ports than a centralized implementation. Because the back-end is clustered, the front-end also performs a steering function aimed at placing dependent instructions within the same back-end cluster, and the cluster assignment logic is also *partitioned* among the clusters. The pipelines implemented in the four front-end clusters work closely in parallel so that instruction blocks fetched during the same cycle advance through the clusters together. As noted earlier, we always assume a single-cycle latency for all signals that pass among clusters. The following subsections describe the approach taken in each of the stages of the clustered front-end pipeline.

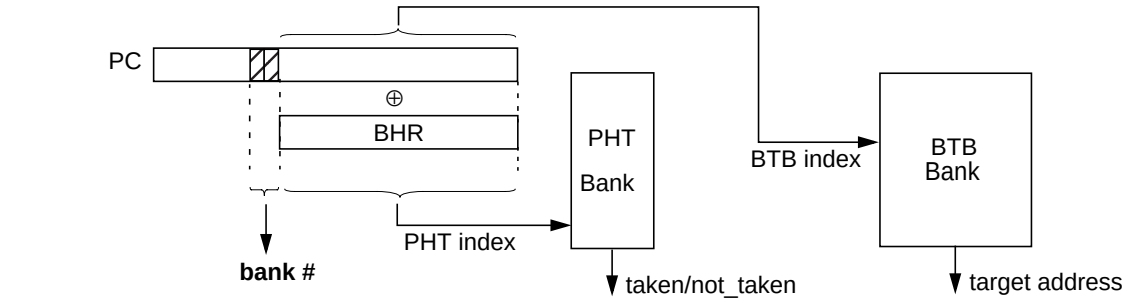
7.2.1 Clustering the Branch Predictor (Stage 1)

As noted before, the branch predictor is part of one of the critical hardware loops [13] of any superscalar processor: the fetch address generation. The fetch address must be generated in a single clock cycle to avoid pipeline bubbles. Thus, the predictor size - hence its accuracy - is greatly limited by cycle time. Clustering the branch predictor may enable larger predictors without increasing their latency, or conversely it may enable faster clocks without losing accuracy.

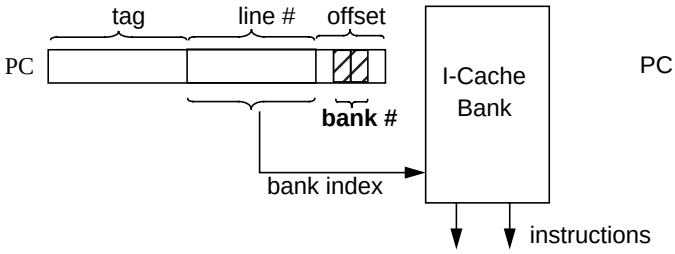
Branch prediction in a clustered front-end introduces new challenges. As with many clustered units, there are two choices for implementing the branch predictor: replicate resources and have each cluster make the same prediction in parallel, or distribute resources and communicate results from one of the clusters to the others. The advantage of replication is that the communication latency between clusters is avoided. The disadvantage is that overall the predictor consumes four times the area with no additional prediction accuracy. The advantage of partitioning and distributing the branch predictor is that it is effectively four times bigger, but prediction results must be communicated to all the clusters from the one that is making any given prediction.

We chose to develop a partitioned design and to integrate it into the pipeline in such a way that the communication delay leads to negligible performance degradation. Our approach, which applies to both the BTB and branch direction predictors, places the predictors at the beginning of the pipeline and divides the predictors into four banks, one per cluster, interleaved by certain bits of the program counter. The program counter is replicated in all the clusters, so the time a predictor bank of size S takes to update its local copy is exactly the same as a centralized predictor of the same size S, i.e., a centralized predictor and a clustered one that is four times bigger fit into the same cycle time.

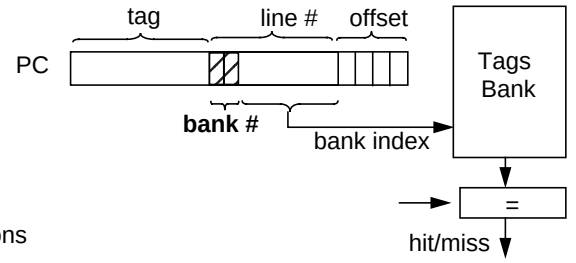
Each cycle, the prediction bank in one of the clusters is active, depending on the bank selection bits taken from the current fetch address. The active predictor is accessed and its prediction must be broadcast to the other clusters so that the replicated PCs are kept consistent. Because this communication is assumed to take a full clock cycle, the rest of PCs are updated one cycle later. While the active cluster does not change, the predictor may keep producing one prediction per cycle. However, when the predictor changes from one cluster to another, a one



(a) Branch predictor: high-order bits interleaving



(b) I-cache instructions: low-order bits interleaving



(c) I-cache tags: high-order bits interleaving

Figure 7-3: Branch predictor (gshare) and instruction cache interleaving

cycle bubble is inserted into the pipeline, because the next prediction cannot start until it can determine the next fetch address. Therefore, the BTB and the predictor tables must be interleaved using some *high order* bits of the address (figure 7-3a) because then, if the code has good spatial locality, it is likely that the same predictor bank is active for many consecutive cycles without creating many bubbles.

Bank switch occurrences could be further reduced by interleaving on even higher order bits of the address, beyond the bits of the index. However, if too high order bits are chosen, a particular code may then run entirely on a single or few predictors, possibly increasing conflicts and losing accuracy. Therefore, the choice of interleaving bits must trade cluster switch frequency for prediction accuracy, which is studied in section 7.3.2. However, for most of our experiments we just assume the scheme outlined in figure 7-3a.

While using the same address to build their index, the predictor and the instruction fetch cannot start at the same time, as it occurs in conventional architectures (figure 7-4a), because it takes one cycle to broadcast the PC to all clusters. In our clustered design the actual instruction fetch stage is delayed by one cycle (stage 2) so that the communication (shown as Bcast in figure 7-4b) may be pipelined smoothly. Note that the prediction stage only depends on the *local_PC*, not the *globally broadcast PC*, unless there is a bank predictor switch, then it depends on the broadcast PC, and a bubble is inserted in the pipeline (see figure 7-4c). Figure 7-5 depicts the pipelines of the baseline (centralized) and clustered front-end architectures.

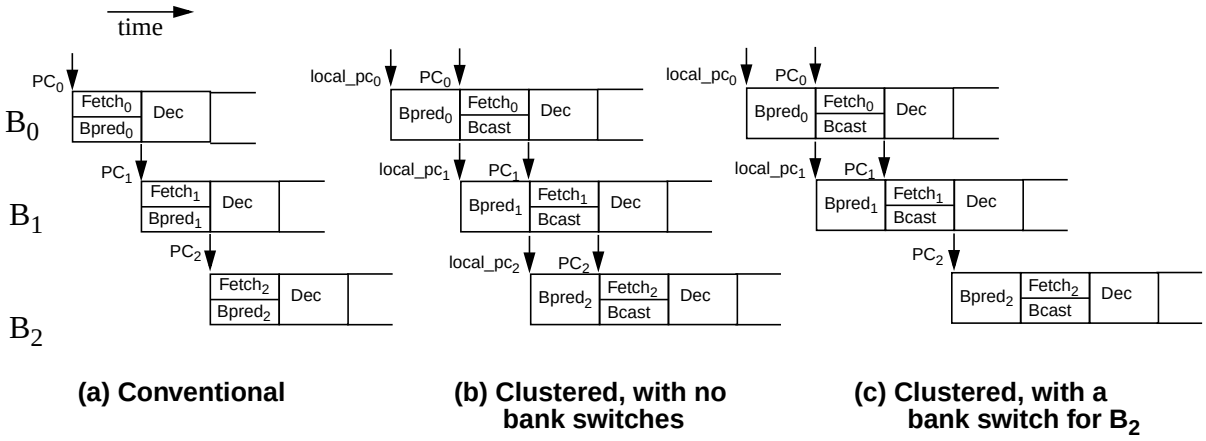


Figure 7-4: Pipeline timing of conventional and clustered front-ends. The diagram shows the fetch of 3 blocks of instructions B0, B1 and B2, at addresses PC_0, PC_1 and PC_2 respectively

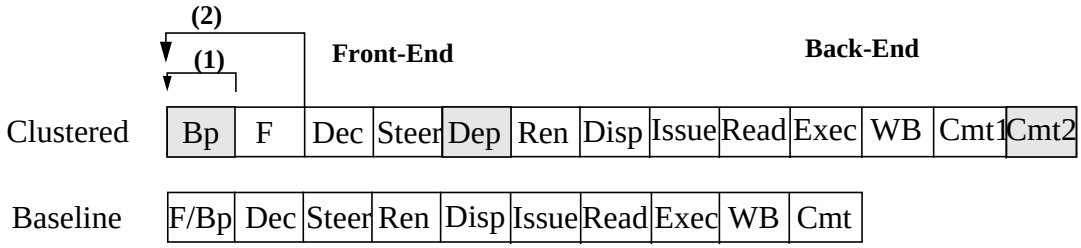


Figure 7-5: Clustered and Baseline (centralized) integer pipelines. Fetch address generation loop is (1) in both pipelines, but it becomes (2) in the clustered one when switching predictor cluster, producing a bubble.

7.2.2 Clustering the I-Cache (stage 2)

Assuming that the predictor is clustered as described above, clustering the I-cache is straightforward and becomes the natural choice for reducing structure size and wire length. Our proposed I-cache is partitioned into banks, which are distributed among the clusters. Each bank is accessed with the local copy of the PC. Hence, assuming equally optimized cache layouts, the fetch time of the clustered cache is exactly the same as that of a centralized cache with one fourth its size. To associate one decoder to each I-cache bank, cache banks must be block-interleaved, i.e. interleaved by some low order address bits of the line offset (see figure 7-3b), so that each bank delivers a portion of the cache line that includes one or more adjacent instructions (2 instructions, on our 8-way 4-cluster example).

The I-cache tag arrays are also clustered to get one tag bank next to each PC. No matter whether the tags are interleaved by the high order or the low order bits of the cache line index, only one bank is activated each cycle. However, for those designs that allow fetching instruction blocks spanning two consecutive lines, interleaving by the high order bits will be a better choice (see figure 7-3c), because it will increase the likelihood that consecutive lines map to the same tag bank. The active tag cluster generates a hit/miss signal and forwards it to the other clusters.

Due to the broadcast delay, this signal is not available to the other clusters until one cycle later. Hence, in case of a cache miss the instructions fetched during the current and the following cycles must be squashed.

An alternative design could interleave the branch predictor and cache tags with the same address bits. Then the active predictor and tag bank would be always in the same cluster, and the tag check could be anticipated during the prediction (stage 1), so that the hit/miss signal could be delivered to all clusters by the end of the fetch stage. This approach would be especially appropriate for set associative caches, where the way selection bits are sent along with the hit/miss signal.

7.2.3 Decode (stage 3)

Each cluster decodes 2 instructions per cycle in stage 3. In addition, the I-cache hit/miss signal is broadcast during this stage. To check dependences for renaming, the destination register designators of all eight possible instructions are broadcast to the other clusters, which takes one clock cycle, during stage 4.

7.2.4 Clustering the Steering Logic (stage 4) and Broadcast of Cluster Assignments (stage 5)

The steering logic assigns each instruction to one back-end cluster, where it will be later steered during dispatch. Note that in our assumed architecture, this task must take place before renaming. The rename logic allocates a free physical register for each instruction that produces a result. However, if the register file is distributed, there is a separate free list for each back-end cluster, and the rename logic allocates one physical register only from the free list of the cluster assigned to the instruction, hence this cluster must be known before renaming.

As noted before, dependence-based steering schemes minimize inter-cluster communications and has proven the most effective scheme for architectures with distributed structures in the back-end. However, it involves a serial task because the steering decisions for one instruction depend on the assignments made for previous instructions. Hence, fitting the delay of the steering logic into a single clock cycle may be difficult, but spreading it into more than one cycle may introduce bubbles into the pipeline, because the outcome for a given block of instructions is needed by the next one. The problem of reducing the total latency of the steering logic may be handled by parallelizing the steering task and partitioning it into clusters, in the following way.

Let us first consider a naive approach that partitions the steering logic into four clusters, and lets each cluster make the steering decisions for its two local instructions, independently of the assignments in other clusters, during stage 4. Each cluster performs just one fourth of the total task, with a great reduction of the total latency. The cluster assignments produced in each cluster are broadcast to the other clusters during the next cycle (stage 5), so they are available to the steering logic of all clusters after two cycles. The renaming, which needs to know the assignments of all eight instructions, may take place in stage 6, and the actual steering of

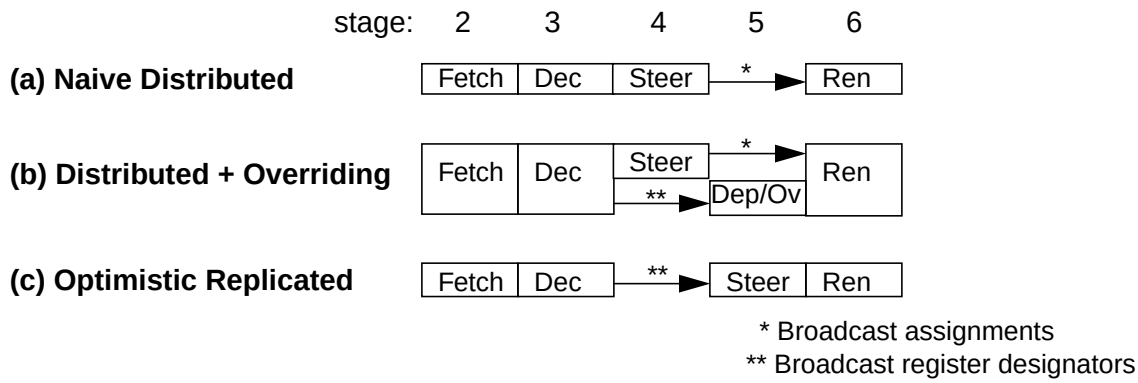


Figure 7-6: Pipeline diagrams for three clustered steering logic schemes

instructions to clusters and insertion into the issue queues occurs in stage 7 (figure 7-6a). The main drawback of this scheme is that the local steering logic in one cluster not only ignores the assignments made by other clusters for instructions of the same block but it also ignores the assignments made during the previous cycle because they are still being broadcast. In other words, the steering logic uses two-cycle outdated assignment information. As a consequence, some instructions get steered to clusters different to the producers of their source operands. Program correctness is still guaranteed because the required copy instructions are generated during the rename stage according to correct assignment information, which is already known at that time. However, we found that using outdated information to make steering decisions has a significant impact on performance (see section 7.3.3) because the distance between producers and consumers is usually short, which causes many extra communications.

Our approach makes a similar partitioning of the steering logic, using partially outdated assignment information but, unlike the naive scheme, the produced cluster assignments are considered provisional. Then, a register dependence analysis is performed in parallel, to check if an assigned instruction depends on one of the instructions whose assignments were ignored by the steering logic. If the producer of a value is found among the previous instructions then, when all assignments are known at the end of stage 5, the assignment of the producer overrides the provisional assignment (figure 7-6b), so that the instruction is steered to the same cluster as the producer.

In more detail, each cluster must check all possible dependences between its two local instructions and the preceding instructions belonging to the same or the previous fetched block (see figure 7-7). The dependence analysis compares the source and destination register names of these instructions. These names are first produced by the decoders in various clusters in stage 3, then they are broadcast to all clusters during stage 4, and the dependence analysis takes place during stage 5, in parallel with the broadcast of the assignments. At the end of stage 5, for each instruction, a multiplexer determines its final assignment by choosing among its provisional assignment or the assignment of some precedent instruction, depending on the results of the dependence check logic.

Finally, for comparison purposes, we also consider an idealized approach to clustering the steering logic that uses perfectly updated assignment information and thus it avoids

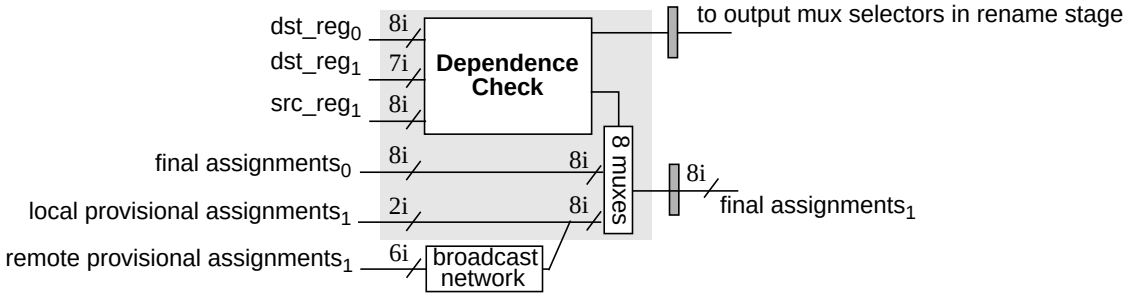


Figure 7-7: Block diagram of the Dependence Check and Overriding (stage 5) for one cluster. Signals with subscript 1 belong to the current block of 8 instructions. Signals with subscript 0 belong to the previous block

performance degradation. This scheme replicates the steering logic in all clusters, so that each cluster makes identical cluster assignments for all eight instructions. Since the required source register designators are broadcast in stage 4, the cluster assignments take place in stage 5 (figure 7-6c). Note that this scheme is optimistic because it assumes that steering eight instructions can be done in a single cycle.

As described before, the steering logic uses DCOUNT workload counters to measure workload balance. These counters are updated during the steering stage (stage 4), according to the provisional assignments. Since many of these assignments are later overridden the workload imbalance measured by the steering logic suffers a loss of accuracy that translates into an imbalance increase. However, we found that by simply tuning the imbalance threshold used in the steering heuristic, the optimal trade-off between communication and imbalance is restored with almost no performance loss. In more detail, the optimal imbalance thresholds were found experimentally to be 32 and 16 for steering logics with updated and partially outdated information, respectively.

7.2.5 Clustering the Rename Logic (stage 6)

Parallelizing the rename logic is not a trivial task. Advanced techniques for parallelizing the rename logic have been proposed elsewhere [66]. In this work we assume a simple scheme that replicates the rename table and the free lists in all clusters, and keeps them consistent by making identical allocation and renaming operations for all eight destination registers in every cluster. Although such a replication has an area cost, there is a reduction in the number of read ports of the map table by a factor of four, because only the sources of the local instructions are renamed in each cluster (two instructions in the 4-by-2 microarchitecture we are considering).

However, to correctly rename these source registers, the rename logic must consider their dependences on instructions being renamed in all clusters. In addition, to maintain consistent copies of the free lists and rename tables in all clusters, the rename logic in each cluster must know the final assignments of the destination registers of all eight instructions. Hence, the dependence check logic and the overriding muxes in stage 5 (see figure 7-7) must be replicated

to produce identical results in all clusters. Note that some of the dependence check results are used to drive the output multiplexers of both the assignment override and the rename logic.

The rename logic in each cluster renames the local instructions (two instructions in the 4-by-2 microarchitecture we are considering), but for correct operation it must check dependences against instructions being renamed in all clusters. Therefore, the free-list and the rename table are replicated in all clusters and they are kept consistent by making identical allocation and renaming operations for all eight destination registers in every cluster. Although such a replication has an area cost, there is a reduction in the number of read ports of the map table by a factor of four, because only the sources of the two local instructions are renamed in each cluster.

7.2.6 Dispatch (stage 7)

In this stage, the renamed instructions are inserted into the appropriate issue queues in the back-end. We assume this operation takes one cycle, because it involves communication from any cluster to any issue queue, which is comparable to other inter-cluster communications.

The dispatch logic steers instructions from any front-end cluster to any back-end cluster. Each cycle, eight regular instructions plus required copies may be generated and steered. To avoid excessive hardware complexity at this point we constrain each issue queue to receive at most eight instructions per cycle. An issue queue may be the target for more than eight instructions only if one regular instruction requires two copies and both are sent to the same cluster. In this case, the pipeline is stalled for one cycle and the two copies are dispatched in consecutive cycles. However, due to the dependence-based nature of the proposed steering heuristic, this case occurs very rarely. We have experimentally observed less than two cases per million cycles.

The renamed instructions are also inserted into the Reorder Buffer (ROB) during the dispatch stage. The ROB is distributed in the following way. Each cluster has a local ROB that is managed as a FIFO queue. After renaming, two instructions from each cluster are inserted into their corresponding local ROBs. To simplify in-order instruction commit, insertions in all four buffers are coordinated, i.e. 2 entries in each buffer are allocated to every block of instructions fetched in the same cycle, even if the group has fewer than 8 instructions (in this case, some entries are left empty).

The clustered reorder buffer (ROB) must accommodate register copy instructions that may get generated during renaming. Because a copy instruction shares most of its entry fields with its source instruction, an implementation of the ROB could save space by storing the (possible) copy and the source instruction in a single extended entry with optional fields.

7.2.7 Back-End Timing and Commit

We complete our description of the baseline back-end with a summary of the assumed pipeline timing to be used during performance evaluation in the next section. We assume that Instruction Issue, Register Read, ALU Execute, and Writeback are similar to those of a conventional superscalar architecture and take one cycle each.

Instruction commit is assumed to take two pipeline stages as follows. Completed instructions commit in-order as they reach the head of their ROB. To maintain synchronized ROBs, up to eight instructions at the two head entries of all four ROBs must be able to commit at once. To ensure that all eight instructions are complete, the four clusters broadcast the *done* status bits of their two head ROB entries to the other clusters, which takes an additional cycle. This has little impact on performance as it is out of the critical path.

7.3 Microarchitecture Evaluation

In this section, we evaluate the efficiency of the proposed eight-issue four-clustered front-end by comparing it with a baseline processor that uses a conventional centralized front-end to drive the same four-clustered back-end. Note that a strict performance comparison is difficult because the primary motivation for a clustered microarchitecture is to manage on-chip wire delays and permit a very fast clock cycle in future chip technologies. Given the cycle-level simulation being performed, we are only able to measure performance as instructions per cycle (IPC). Hence, the expected advantage in cycle time is not measured. In particular, our IPC performance goal becomes one of matching the performance of using a centralized front-end.

For the experiments in this chapter, we simulated the SpecInt95 benchmark suite, as described in chapter 2. The branch predictor is assumed to operate in a single clock cycle, so we simulate a moderately sized gshare/bimodal hybrid predictor. The bimodal has a 2K-entry table of 2-bit counters; the gshare has a 16K-entry PHT table; and the meta predictor has a 2K-entry table of 2-bit counters. The rest of the architectural parameters are summarized in table 2-1.

7.3.1 Performance

Figure 7-8 compares performance, measured as committed instructions per cycle (IPC) of the clustered front-end microarchitecture to that of the baseline microarchitecture. The graph shows that the IPC of the clustered approach is, on average, 4% lower than that of the baseline. This result is quite uniform for all benchmarks, and has three causes: about 1% is due to switching the predictor banks, another 2.7% is caused by the one-cycle increase of branch misprediction penalty, due to the additional assignment broadcast stage (labelled Dep in figure 7-5), and the rest is caused by the partitioned steering.

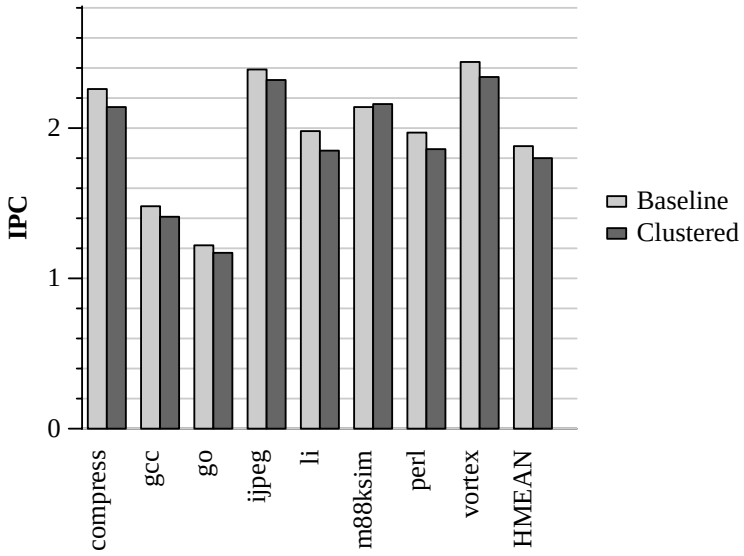


Figure 7-8: IPC of the baseline and the clustered front-end schemes

As discussed in section 7.2.1, the delay of a centralized predictor is the same than a clustered one with four times its size. As an experiment to at least partially account for the clock cycle advantages of the distributed front-end, we performed a second set of simulations where the centralized branch predictor was assumed to have the same size as the predictor in each of the four clusters (hence the distributed predictor is four times as large).

In addition, all the processing tasks between Instruction Fetch and Dispatch are the same for both architectures. The fact that the clustered one has an additional stage does not mean that it has more levels of logic between these two pipeline points, but these levels of logic are spread differently among stages. Therefore, in these experiments it would be fair that if we assume the same cycle time in both cases, we assume also that both architectures have the same latency between these two pipeline stages, by adding one stage to the centralized front-end. Of course, the resulting IPC has not a direct meaning, but it is rather the speed-up the significant metric to be used in this case. When this is done, the clustered front-end shows a 1% average IPC speed-up over the centralized one. Although the speed-up is small, we believe that we have been quite conservative, and we have not factored in all the advantages of the clustered design.

7.3.2 Impact of Partitioning the Branch Predictor

Partitioning the branch predictor produces a one-cycle pipeline bubble each time the cluster that makes the prediction changes. On average, this occurs once every 25 committed instructions for SpecInt95. We evaluated the performance impact of the bank switch penalty on a clustered front-end architecture by comparing it to a similar model with a centralized predictor. It is shown in Figure 7-9 that the one-cycle penalty associated with switching the predictor cluster produces an average performance loss of 1%. The two predictors have identical total size but the clustered one is faster. Alternatively, we could compare predictors having equal latency, by making the predictor in each cluster the same size as the centralized one. The results of this experiment showed a 2.4% average performance speed-up for the clustered predictor.

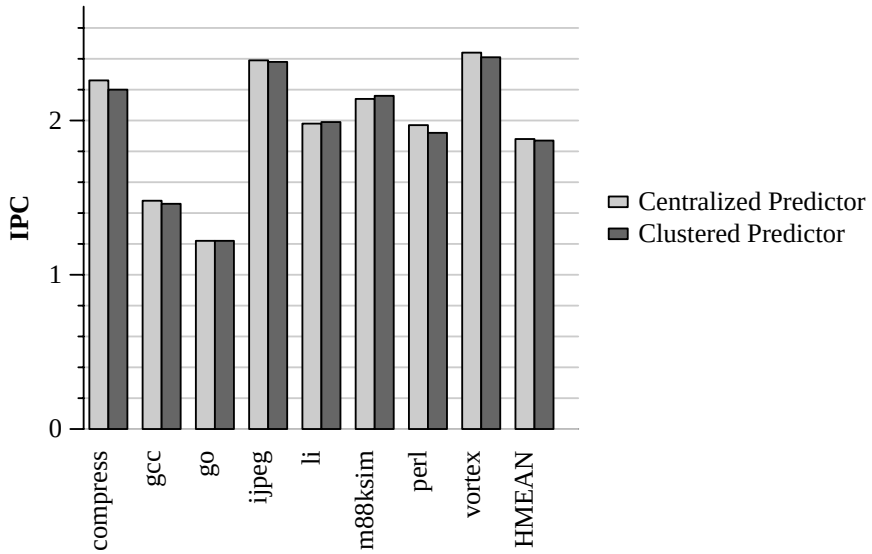


Figure 7-9: Impact of the predictor bank switch penalty. A clustered predictor is compared to a centralized one.

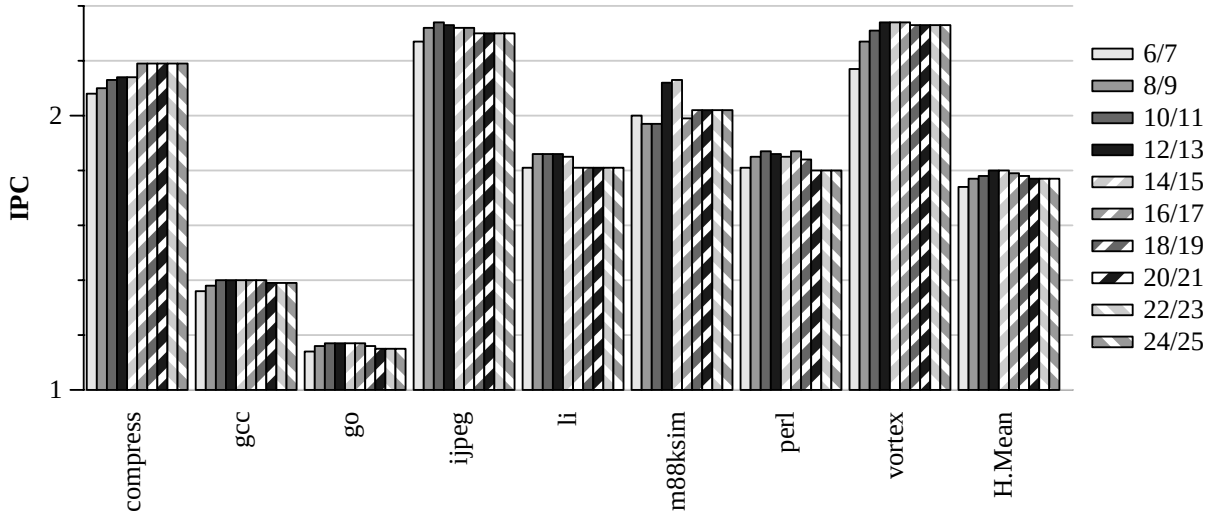


Figure 7-10: Performance impact of the interleaving bits, for a clustered front-end

The interleaving scheme of the default clustered predictor assumed for most experiments (see figure 7-3) interleaves the predictor by the high order bits of the fetch address used to build the index (i.e., the index uses bits 2 to 15, and the default scheme interleaves by bits 14/15). As noted earlier, interleaving by even higher order bits may reduce the amount of bank switching but also may increase aliasing because it tends to concentrate most accesses on the same predictor bank. Conversely, interleaving by lower order bits has the opposite effects. The impact of the predictor interleaving scheme on performance is analyzed in figure 7-10. It is shown that the optimal interleaving differs from one benchmark to another but, overall it stays between bits 12/13 and 16/17, and average performance is maximum for our default interleaving scheme.

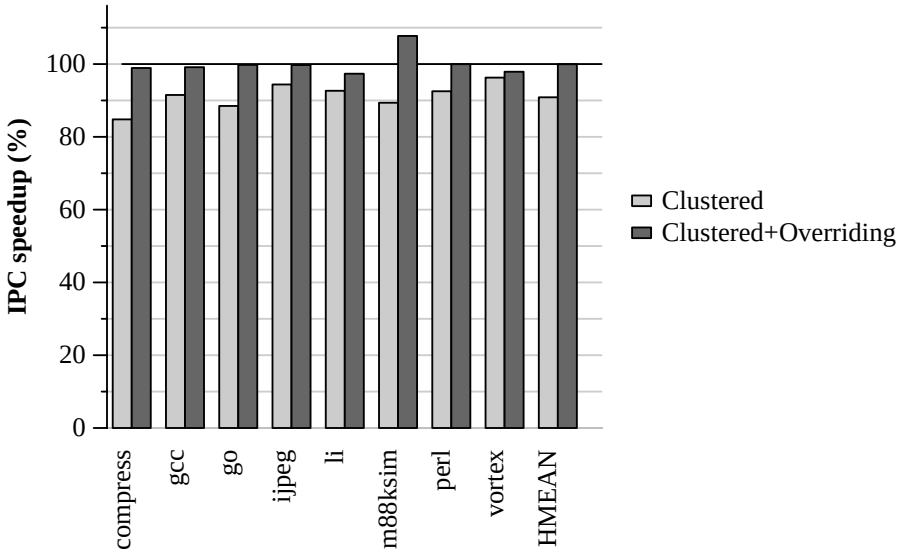


Figure 7-11: Impact of using 2-cycle outdated renaming information by the steering heuristic: two clustered front-ends are compared, a naive scheme and our approach with assignment overriding (speed-ups are relative to a clustered front-end with replicated steering logic that uses updated information)

7.3.3 Impact of Steering with Outdated Renaming Information

Distributing the register renaming function forces steering decisions to be made two cycles before renaming, to have time to broadcast them to all clusters. We evaluated the impact of using outdated renaming information and the effectiveness of the assignment overriding technique by comparing two clustered front-end microarchitectures, with and without overriding, to a model with replicated steering logic that uses totally up-to-date information.

Figure 7-11 shows that using outdated information with the naive scheme, without overriding, produces a 9% average performance loss. Performance loss is due to a drastic increase in inter-cluster data communications between dependent instructions (from 0.18 to 0.43 communications per instruction). However, in our approach with assignment overriding, inter-cluster data communications fall to 0.18 communications per instruction, and the partitioned steering scheme has the same performance as the replicated scheme.

7.4 Conclusions

In this paper, a novel design to fully distribute the processor's front-end is proposed, which extends the advantages of clustering to structures like the I-cache, the branch predictor, the steering logic and the renaming map table.

Several techniques are proposed to partition these structures with the goal of reducing their complexity, and avoiding replication. These techniques minimize the wire delay penalties

caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation, and the cluster assignment.

First, it is shown that the branch predictor latency, which is in the critical path of the fetch address generation loop, may be reduced by partitioning it into clusters, in such a way that cross-structure wire delays are left out of the critical path and converted to cross-cluster communications that may be smoothly pipelined.

Second, it is shown that the latency of a dependence-based steering scheme, which is inherently a serial process that forms a critical hardware loop, may be greatly reduced by partitioning it into clusters. The negative impact of using outdated assignment information is effectively mitigated with an overriding scheme driven by the dependence analysis.

In summary, clustering reduces the latency of these hardware structures and expose wire delays so that they can be dealt with effectively, which results in faster clock rates and translates into significant performance improvements. The schemes proposed in this paper to deal with inter-cluster communications are very effective since they reduce communication penalties to just a 4% IPC degradation, for SpecInt95.

CHAPTER 8

CONCLUSIONS

In this chapter, the main conclusions of this thesis are presented, as well as some open-research areas for future work.

8.1 Conclusions

First, in this thesis, several dependence-based dynamic cluster assignment schemes have been proposed for a clustered superscalar processor.

The first main proposal is a family of slice-based, dynamic cluster assignment schemes. This kind of schemes assign the set of instructions involved in a load or store address calculation (Ld/St slice), or in a branch condition calculation (Br slice), to the same cluster, because these instructions are likely to be critical. First, a dynamic scheme based on a previous "load/store slice" compile-time algorithm is proposed, and it is shown that its major weakness is that it often produces poor workload balance. Then, other more efficient new dynamic slice-based schemes are presented which greatly improve the ability to keep the workload balanced among clusters.

All of these algorithms are evaluated for a cost-effective architecture which is a two-cluster architecture that results of extending the FP unit of a conventional superscalar to execute simple integer operations. It is shown that the dynamic load/store slice scheme substantially outperforms the static one. It is also shown that our best slice-based scheme outperforms a state-of-the-art previous dynamic proposal because it generates a significantly lower number of inter-cluster communications. Our results also prove that a cost-effective architecture with our cluster assignment schemes substantially outperforms a conventional one when running the SpecInt95 benchmarks, because it can profitably exploit idle FP resources to speed-up integer programs with minimal hardware support and no ISA change. It is also shown that this cost-effective

architecture also outperforms a conventional architecture when running the SpecFP95 benchmarks, even though the FP-cluster datapath is shared by integer and FP instructions.

The second main proposal is a family of new dynamic schemes (referred to as RMB) that assign instructions to clusters in a per-instruction basis, based on prior assignment of the source register producers, the cluster location of the source physical registers, and the workload of clusters. Compared to the slice-based algorithms, this class of schemes have lower implementation cost and are more flexible to keep the workload balanced since they operate at a finer granularity. The proposed RMB schemes follow a primary and secondary criteria that specifically address the goals of minimizing communications and maximizing workload balance respectively. Our proposal includes a precise definition for the workload balance among N clusters, as well as a hardware mechanism to estimate it.

The experiments show that the proposed AR-Priority RMB steering scheme is more effective than all the existing slice-based schemes, and it significantly outperforms the best dynamic schemes previously proposed in the literature, because it achieves the best trade-off between communications and workload balance. It is also shown that this cluster assignment scheme scales better than other RMB proposals, with growing number of clusters and communication latency, because it avoids spreading blindly the instructions among clusters thus producing less communications. Two more conclusions are extracted from our evaluation: on the one hand, a clustered architecture is quite sensitive to the inter-cluster communication latency, which emphasizes the importance of reducing communications with appropriate steering decisions. On the other hand, a clustered architecture with the AR-Priority RMB assignment scheme is hardly sensitive to the cluster interconnect bandwidth because it has a low bandwidth demand, which suggests the feasibility of efficient cluster interconnects based on simple hardware.

The third main proposal of this thesis is to reduce the penalties of wire delays through the prediction of the communicated values. It is motivated by the increasing penalty of wire delays in future microprocessors. The proposed strategy is applied to inter-cluster communications, in a clustered superscalar architecture. The proposal includes a novel cluster assignment scheme (VPB) that avoids data dependence constraints on the assignment algorithm when a value is going to be predicted and there is a potential for improving the workload balance. This algorithm exploits the less dense data dependence graph that results from predicting values to achieve a better workload balance.

It is shown that value prediction removes communications even for previously proposed steering schemes not specially designed to exploit value prediction. However, performance is higher with VPB cluster assignment because it exploits the predictability of values to improve the workload balance. A simple stride value predictor, together with VPB steering, removes on average 50% of the communications, and reduces substantially the workload imbalance, which translates into an average 14% IPC speedup for a four-clustered architecture. In contrast, an identical value predictor achieves only a 3% speedup for a centralized 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, so this technique may produce even better improvements in future technologies, as wire

delay - and hence communication latency - increases. On the other hand, it is also shown that the performance improvement of value prediction is quite sensitive to misprediction penalty, although it is less sensitive to the predictor table size, for the considered set of benchmarks and table sizes.

The fourth proposal is motivated by the observation that a clustered architecture (with a good steering scheme, such as the AR-Priority RMB) is hardly sensitive to the cluster interconnect bandwidth. Therefore, cluster interconnects can be implemented with simple hardware and with very low complexity impact on the register files, the issue windows and the bypasses, which results in short cycle times and low power consumption. Several cost-effective point-to-point interconnects are proposed, both synchronous (a ring) and partially asynchronous (a ring, a mesh and a torus). 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 instance, for a synchronous ring interconnect, the router hardware is as simple that it only requires five registers and three multiplexers per cluster. In addition, a new topology-aware improvement to the cluster assignment scheme is proposed to reduce the distance (and latency) of inter-cluster communications.

It is shown that the simple point-to-point interconnects, together with our effective steering schemes, achieve much better performance than bus-based interconnects. Besides, the former do not require a centralized arbitration to access the transmission medium. It is also shown that a partially asynchronous ring performs better than the synchronous one at the expense of some additional cost/complexity due to the additional queue required per cluster, although a tiny queue would practically never overflow. Regarding the connectivity and bandwidth, we extract the following two conclusions. First, as expected, interconnects with higher connectivity perform better because they have shorter communication latency. However, point-to-point partially asynchronous interconnects with moderate connectivity/complexity perform close to an idealized crossbar with unlimited bandwidth and all nodes at one-cycle distance: e.g., a four-cluster ring performs within 2% of the ideal, and an eight-cluster torus performs within 4% of the ideal. Second, despite the low hardware requirements of partially asynchronous interconnects, they achieve a performance close (within 1%) to an equivalent idealized interconnect with unlimited bandwidth and number of write ports to the register files.

Finally, a novel design to fully distribute the processor’s front-end is proposed, which extends the advantages of clustering to structures like the I-cache, the branch predictor, the steering logic and the renaming map table. Several techniques are proposed to partition these structures with the goal of reducing their complexity, and avoiding replication. These techniques minimize the wire delay penalties caused by broadcasting recursive dependences in two critical hardware loops: the fetch address generation, and the cluster assignment. First, it is shown that the branch predictor latency, which is in the critical path of the fetch address generation loop, may be reduced by partitioning it into clusters, in such a way that cross-structure wire delays are left out of the critical path and converted to cross-cluster communications that may be smoothly pipelined. Second, it is shown that the latency of a dependence-based steering scheme, which is inherently a serial process that forms a critical hardware loop, may be greatly reduced by partitioning it into clusters. The negative impact of using outdated assignment information is effectively mitigated with an overriding scheme driven by the dependence analysis.

In summary, clustering reduces the latency of these hardware structures and expose wire delays so that they can be dealt with effectively, which results in shorter pipelines and/or faster clock rates and translates into significant performance improvements. Ignoring these performance benefits, and assuming a full clock cycle for any cross-cluster communication, partitioning the front-end produces just a 4% IPC loss for SpecInt95. This result is quite uniform for all benchmarks, and may be broken into three components. About 1.65% IPC loss is caused by the one-cycle increase in branch misprediction latency, which results from adding a stage to broadcast the cluster assignments. About 1% is due to the clustered branch predictor, because of the bubble generated when switching the predictor banks, and the remaining 1.35% is caused by using outdated assignment information in the partitioned steering.

8.2 Open-Research Areas

For a clustered architecture, inter-cluster communication latency is a major source of performance loss. The baseline architecture assumed in this thesis uses a simple explicit copy mechanism to transfer values from one register file to another, and these copy instructions are handled in the same way as regular instructions. Though simple, this mechanism has two important drawbacks: first, it consumes issue queue entries and issue bandwidth; second, copy instructions augment the data dependence graph with an additional node that increases the length of the dependence chain, and may contribute to the length of the critical path of execution. As suggested in section 2.1.3, the first problem may be solved with minimal hardware cost by using separate issue queues and register file read ports for copy instructions. However, the second problem still remains: copy instructions increase the total communication latency by one cycle, which is spent in the wake-up and selection task of the copy instruction itself. Therefore, further research on improved register copy mechanisms with the goal of avoiding the copy issue cycle penalty is ongoing.

Along this thesis we deliberately obviate the question of which is the optimal cluster configuration for a given technology scenario, and a given design goal. Our approach considers configurations with homogeneous clusters, and with a moderate number of clusters (2 or 4) and issue width per cluster (2 or 4). However, other configurations could be explored. On the one hand, whereas increasing the amount of execution resources per cluster has a positive effect on IPC, it also increases the delay of some critical hardware loops, such as the wakeup and select task or the execute and bypass loop. Since both effects have opposite directions, an accurate delay estimation could be useful to study the impact on performance of many design trade-offs between parallelism and complexity. On the other hand, it is ineffective to replicate in all clusters execution resources for complex operations that are seldom used. Thus, exploring configurations with heterogeneous clusters could be an interesting approach.

The design of the cache hierarchy for a clustered architecture is another important open research area. There are two main architectural components that should be studied: L1 caches and disambiguation hardware. Current technology projections anticipate that first level data caches will tend to reduce their size to keep latency low, although it will come at the cost of a higher miss ratio. With clustered architectures, a partitioned cache design may seek to achieve

both objectives, a reduced latency and a large cache size. There exist already some proposals to design a clustered L1 data cache. However, there are less work done that focuses on the complexity of the dynamic disambiguation hardware. Although the issues of partitioning the disambiguation logic and data caches are independent of other architectural features, it is likely that both may take advantage of the instruction distribution mechanisms already included in a clustered architecture, so it may be viewed as a further extension of the clustered approach.

This thesis has shown the benefits of value prediction to reduce inter-cluster communications in a clustered architecture. However, our approach focused on a rather simple stride based value prediction scheme. It is likely that even better performance could be achieved with more clever prediction schemes that are both more accurate and more selective, to focus on those values that are more predictable, and have the highest potential for reducing the critical path of execution.

BIBLIOGRAPHY

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proc. of the 27th Ann. Intl. Symp. on Computer Architecture*, pp. 248-259, June 2000.

[2] V. Agarwal, S.W. Keckler, D. Burger. The Effect of Technology Scaling on Microarchitectural Structures. Tech. Report #TR2000-02, Dept. of Computer Sciences, The Univ. of Texas at Austin, May 2001.

[3] A. Aggarwal and M. Franklin. An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software*, pp. 172-179, November 2001.

[4] A. Aggarwal and M. Franklin. Hierarchical Interconnects for On-chip Clustering. In *Proc. of the Intl. Parallel and Distributed Processing Symposium*, pp. 63-70, April 2002.

[5] A. Aggarwal and M. Franklin. Instruction Replication: Reducing Delays due to Inter-PE Communication Latency. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 46-55, October 2003.

[6] H. Akkary and M.A. Driscoll, A Dynamic Multithreading Processor. In *Proc. of the 31st. Ann. Intl. Symp. on Microarchitecture*, pp. 226-236, December 1998.

[7] R.I. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proc. of the 28th Ann. Intl. Symp. on Computer Architecture*, pp. 218-229, July 2001.

[8] R. Balasubramonian, S. Dwarkadas and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In *Proc. of the 30th. Ann. Intl. Symp. on Computer Architecture*, pp. 275-286, June 2003.

[9] S. Banerjia. *Instruction Scheduling and Fetch Mechanisms for Clustered VLIW Processors*. PhD thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, 1998.

[10] A. Baniasadi, and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proc. of the 33rd. Ann. Intl. Symp. on Microarchitecture*, pp. 337-347, December 2000.

[11] R. Bhargava, and L.K. John. Improving Dynamic Cluster Assignment for Clustered Trace Cache processors. In *Proc. of the 30th. Ann. Intl. Symp. on Computer Architecture*, pp. 264-274, June 2003.

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

[13] E. Borch, E. Tune, S. Manne and J. Emer. Loose Loops Sink Chips. In *Proc. of the 8th. Intl. Symp. on High-Performance Computer Architecture*, pp. 270-281, February 2002.

[14] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4): 23-29, July-August 1999.

[15] 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.

[16] R. Canal, J.-M. Parcerisa, A. Gonzalez, A Cost-Effective Clustered Architecture, in *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 160-168, October 1999.

[17] R. Canal, J-M. Parcerisa, A. González. Dynamic Cluster Assignment Mechanisms. In *Proc. of the 6th. Intl. Symp. on High-Performance Computer Architecture*, pp. 132-142, January 2000.

[18] R. Canal, J-M. Parcerisa, and A. González. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, Kluwer Academic/Plenum Publishers, 29 (1): 59-79, February 2001

[19] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proc. of the 25th. Ann. Intl. Symp. on Microarchitecture*, pp. 292-300, December 1992.

[20] R. P. Colwell, R. P. Nix, J.J. O'Donnell, D.P. Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proc. of the 2nd. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[21] J.-L.Cruz, A.González, M. Valero and N.P. Topham. Multiple-Banked Register File Architectures. In *Proc. of the 27th. Intl. Symp. on Computer Architecture*, pp. 316-324, June 2000.

[22] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Tech. Report HPL-98-13, HP Labs, January 1998.

[23] J. Duato, S. Yalamanchili, L. Ni, *Interconnection Networks, An Engineering Approach*, IEEE Computer Society Press, 1997.

[24] 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. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 109-121, 1995.

[25] R.J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4): 547-564, July 1993.

[26] J.R. Ellis. *Bulldog: A Cokpiler for VLIW Architectures*. PhD thesis, Yale University, 1985.

[27] K.I. Farkas, N.P. Jouppi and P. Chow. Register File Design Considerations in Dynamically Scheduled Processors, in *Proc. of the 2nd. Intl. Symp. on High-Performance Computer Architecture*, pp. 40-51, 1996.

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

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

[30] M.M. Fernandes, J.Llosa and N.Topham, Distributed Modulo Scheduling, in *Proc. of the 5th. Intl. Symp. on High-Performance Computer Architecture*, pp. 130-134, January 1999.

[31] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proc. of the 28th. Ann. Intl. Symp. on Computer Architecture*, pp. 74-85, June 2001.

[32] M.J. Flynn, P. Hung, and K. Rudd. Deep-Submicron Microprocessor Design Issues. *IEEE Micro*, 19(4): 11-22, July/August 1999.

[33] M. Franklin and G.S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pp.58-67, May 1992.

[34] M. Franklin. *The Multiscalar Architecture*. PhD. Thesis, Technical Report TR 1196, Computer Sciences Department, University of Winsconsin-Madison, November 1993.

[35] J. Fridman and Z. Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1): 66-76, January-February 2000.

[36] D.H. Friendly, S.J. Patel, and Y.N. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Processors. In *Proc. of the 31st. Ann. Intl. Symp. on Microarchitecture*, pp. 173-181, November 1998.

[37] F.Gabbay and A.Mendelson. Speculative Execution Based on Value Prediction, Tech. Report #1080, Univ. Technion, Israel, 1996.

[38] J.González and A.González. Memory Address Prediction for Data Speculation. Tech. Report #UPC-DAC-1996-50, Univ. Politècnica de Catalunya, Spain, 1996.

[39] J.González and A.González. Speculative Execution Via Address Prediction and Data Prefetching. In *Proc. of the 11th. International Conference on Supercomputing*, pp. 196-203, July 1997.

[40] J.González and A.González. The Potential of Data Value Speculation to Boost ILP. In *Proc. of the 12th. International Conference on Supercomputing*, pp. 21-28, 1998.

[41] L. Gwennap. Intel's MMX Speeds Multimedia Instructions, *Microprocessor Report*, 10(3), March 1996.

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

[43] R. Ho, K.W. Mai, M.A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4): 490-504, April 2001.

[44] M.S. Hrishikesh, N.P. Jouppi, K.I. Farkas, D. Burger, S.W. Keckler, and P. Shivakumar. The Optimal Logic Depth per Pipeline Stage Is 6 to 8 FO4 Inverter Delays. In *Proc. of the 29th Ann. Intl. Symp. on Computer Architecture*, pp. 14-24, May 2002.

[45] *The International Technology Roadmap for Semiconductors*. Semiconductor Industry Association. 1999.

[46] *The International Technology Roadmap for Semiconductors*. Semiconductor Industry Association. 2002.

[47] A. Iyer and D. Marculescu. Power and Performance of Globally Asynchronous Locally Synchronous Processors. In *Proc. of the 29th Ann. Intl. Symp. on Computer Architecture*, pp.158-168, May 2002.

[48] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-Based Next Trace Prediction. In *Proc. of the 30th. Ann. Intl. Symp. on Microarchitecture*, pp. 14-23, December 1997.

[49] J. Johnson. Expansion Caches for Superscalar Processors. Tech. Report CSL-TR-94-630, Computer Science Laboratory, Stanford University, June 1994.

[50] M. Johnson. *Superscalar Microprocessor Design*. Ed. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[51] S.W. Keckler, D. Burger, C.R. Moore, R. Nagarajan, K. Sankaralingam, V. Agarwal, M.S. Hrishikesh, N. Ranganathan, P. Shivakumar. A Wire-Delay Scalable Microprocessor Architecture for High Performance Systems. In *Proc. of the 2003 IEEE Intl. Solid-State Circuits Conference*, paper no. 9.6, February 2003.

[52] G.A. Kemp, and M.Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proc. of Intl. Conf. on Parallel Processing*, pp. 239-246, August 1996.

[53] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24-36, 1999.

[54] H-S.Kim and J.E.Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proc. of the 29th Ann. Intl. Symp. on Computer Architecture*, pp. 71-87, May 2002

[55] K. Krewell. Intel Embraces Multithreading, *Microprocessor Report*, Sept. 2001, pp. 1-2.

[56] C. Lee, M. Potkonjak and W. H. Mangione-Smith, Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, In *Proc. of the 30th IEEE/ACM Intl. Symposium on Microarchitecture*, pp. 330-335, December 1997.

[57] M.H.Lipasti, C.B.Wilkerson, and J.P.Shen. Value Locality and Load Value Prediction. In *Proc. of the 7th. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.138-147, October 1996.

[58] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

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

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

[61] D. Matzke. Will Physical Scalability Sabotage Performance Gains?. *IEEE Computer* 30(9): 37-39, September 1997.

[62] Mediabench Home Page. URL: http://www.cs.ucla.edu/~leec/mediabench/

[63] S. Melvin and Y. Patt. Performance Benefits of Large Atomic Units in Dynamically Scheduled Machines. in *Proc. of the 3rd. Intl. Conference on Supercomputing*, pp. 427-432, June 1989.

[64] R. Nagarajan, K. Sankaralingam, D. Burger and S.W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proc of the 34th. Ann. Intl. Symp. on Microarchitecture*, pp. 40-51, 2001

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

[66] P.S. Oberoi and G.S. Sohi. Parallelism in the Front-End. In *Proc. of the 30th. Ann. Intl. Symp. on Computer Architecture*, pp. 230-240, June 2003

[67] E. Ozer, S. Banerjia, T.M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. in *Proc. of the 31st. Intl. Symp. on Microarchitecture*, pp. 308-315, November 1998.

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

[69] S. Palacharla, N.P. Jouppi, and J.E. Smith. Quantifying the Complexity of Superscalar Processors. Tech. Report CS-TR-961328, Univ. of Wisconsin-Madison, November 1996.

[70] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proc. of the 24th. Intl. Symp. on Computer Architecture*, pp. 206-218, June 1997.

[71] S.Palacharla. *Complexity-Effective Superscalar Processors*. Ph.D. thesis, Univ. of Winsconsin-Madison, 1998.

[72] J.-M. Parcerisa and A. González, The Latency Hiding Effectiveness of Decoupled Access/Execute Processors. In *Proc. of the 24th. Euromicro Conference*, Vasteras, Sweden, pp. 293-300, August 1998.

[73] J.-M. Parcerisa and A. González, Reducing Wire Delay Penalty through Value Prediction, In *Proc. of the 33rd. Intl. Symp. on Microarchitecture*, pp. 317-326, December 2000.

[74] J.-M. Parcerisa, A. González and J.E. Smith. A Clustered Front-End for Superscalar Processors. Tech. Report #UPC-DAC-2002-29, Computer Architecture Dept., Univ. Politècnica de Catalunya, Spain, July 2002.

[75] J.-M. Parcerisa, J. Sahuquillo, A. González and J. Duato. Efficient Interconnects for Clustered Microarchitectures. In *Proc. of the 11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 291-300, September 2002.

[76] S. Patel, M. Evers, and Y. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. In *Proc. of the 25th. Ann. Intl. Symp. on Computer Architetcure*, pp. 262-271, June 1998.

[77] A. Peleg and U. Weiser. Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line. U.S. Patent Number 5,381,533, January 1995.

[78] P. Racunas and Y.P. Patt. Partitioned First-Level Cache Design for Clustered Microarchitectutres. In *Proc. of the 17th Ann. Intl. Conf. on Supercomputing*, pp. 22-31, June 2003

[79] N. Ranganathan and M. Franklin. An Empirical Study of Decentralized ILP Execution Models. *In Proc. of the 8th. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 272-281, October 1998.

[80] N. Ranganathan and M. Franklin. The PEWs microarchitecture: reducing complexity through data-dependence based decentralization. *Microprocessors and Microsystems*, Elsevier B.V., 22(6): 333-343, November 1998.

[81] E. Rotenberg, S. Bennet, and J.E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th. Ann. Intl. Symp. on Microarchitecture*, pp 24-34, December 1996.

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

[83] E. Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. Ph.D. thesis, Univ. of Winsconsin-Madison, 1999.

[84] J. Sánchez and A. González. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In *Proc. of the 33rd. Ann. Intl. Symp. on Microarchitecture*, pp. 124-133, December 2000.

[85] K. Sankaralingam, R. Nagarajan, S.W. Keckler, and D. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. Tech. Report #TR2001-02, Dept. of Computer Sciences, Univ. of Texas, Austin, 2002.

[86] K. Sankaralingam, V.A. Singh, S.W. Keckler and D. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proc. of the 21st. Intl. Conf. on Computer Design*, pp. 170-177, October 2003.

[87] S.S. Sastry, S. Palacharla, and J.E. Smith, Exploiting Idle Floating-Point Resources For Integer Execution, in *Proc. of the Intl. Conf. on Programming Language Design and Implementation*, pp. 118-129, Montreal, June 1998.

[88] Y. Sazeides, S. Vassiliadis, and J.E. Smith. The Performance Potential of Data Speculation and Collapsing. In *Proc. of the 29th Ann. Intl. Symp. on Microarchitecture*, pp. 238-247, December 1996.

[89] A. Seznec, E. Toullec, and O. Rochecouste. Register Write Specialization Register Read Specialization: A Path to Complexity-Effective Wide-Issue Superscalar Processors. In *Proc. of the 35th. Ann. Intl. Symp. on Microarchitecture*, pp. 383-394, November 2002.

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

[91] J.E. Smith and G.S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609-1624, December 1995.

[92] J.E. Smith. Instruction-Level Distributed Processing. *Computer*, IEEE Computer Society, 34(4): 59-65, April 2001.

[93] G.S. Sohi, S.E. Breach and T.N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd. Intl. Symp. on Computer Architecture*, pp. 414-425, June 1995.

[94] S.P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5): 8-17, October 1994.

[95] E. Sprangle, D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proc. of the 29th Ann. Intl. Symp. on Computer Architecture*, pp. 25-34, May 2002.

[96] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strensky, P.G. Emma. Optimizing Pipelines for Power and Performance. In *Proc. of the 35th. Ann. Intl. Symp. on Microarchitecture*, pp. 333-344, November 2002.

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

[98] J. Stark, M.D. Brown and Y.N. Patt. On Pipelining Dynamic Instruction Scheduling Logic. In *Proc. of the 33th. Ann. Intl. Symp. on Microarchitecture*, pp. 57-66, Dec. 2000

[99] J.M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, *POWER4 System Microarchitecture*, Technical white paper, IBM server group web site, October 2001

[100] A. Terechko, E.L. Thenaff, M. Garg, J.van Eijndhoven, and H. Corporaal. Inter-cluster Communication Models for Clustered VLIW Processors. In *Proc. of the 9th. Intl. Symp. on High-Performance Computer Architecture*, 2003.

[101] Texas Instrument Inc. TMS320C62x/67x CPU and Instruction Set Reference Guide, 1998.

[102] T.N. Theis. The Future of Interconnection Technology. *IBM Journal of Research and Development*, 44(3):379-389, May 2000.

[103] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25-33, January 1967.

[104] N.P. Topham, A. Rawsthorne, C.E. McLean, M.J.R.G. Mewissen and P. Bird. Compiling and Optimizing for Decoupled Architectures. In *Proc. of the Supercomputing '95*, San Diego, December 1995.

[105] M. Tremblay, J. Chan, S. Chaundrhy, A.W. Conigliaro, S.S. Tse, The MAJC Architecture: A Synthesis of Parallelism and Scalability, *IEEE Micro* 20(6): 12-25, November/December 2000.

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

[107] E. Tune, D. Liang, D.M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proc. of the 7th. Intl. Symp. on High-Performance Computer Architecture*, pp. 185-195, January 2001.

[108] S. Vajapeyam and T. Mitra, Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proc. of the 24th. Intl. Symp. on Computer Architecture*, pp. 1-12, June 1997.

[109] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, pp. 86-93, September 1997.

[110] K.C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2): 28-41, April 1996.

[111] A. Yoaz, M. Erez, R. Ronnen and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proc. of the 26th. Ann. Intl. Symp. on Computer Architecture*, pp. 42-53, 1999

[112] V. Zyuban. *Inherently Lower-Power High-Performance Superscalar Architectures*, Ph.D. thesis, Univ. of Notre Dame, January 2000.

[113] V.V.Zyuban and P.M. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers* 50(3): 268-285, March 2001

LIST OF FIGURES

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

LIST OF TABLES

Chapter 2

Chapter 3

Chapter 6