12-7-2007

Intra-level Incomplete Bypassing: Achieving Performance and Power Efficiency

Eric P. Villasenor
Purdue University - Main Campus, evillase@purdue.edu

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis Acceptance

This is to certify that the thesis prepared

By Eric Villasenor

Entitled

Intra-cluster Incomplete Bypassing: Achieving Performance and Energy Efficiency

Complies with University regulations and meets the standards of the Graduate School for originality and quality

For the degree of Master of Science in Electrical and Computer Engineering

Final examining committee members

M. S. Thottethodi, Chair	
T. N. Vijaykumar	
Y. Lu	

Approved by Major Professor(s): M. S. Thottethodi

Approved by Head of Graduate Program: M. J. T. Smith

Date of Graduate Program Head's Approval: 12/06/07

INTRA-LEVEL INCOMPLETE BYPASSING:
ACHIEVING PERFORMANCE AND POWER EFFICIENCY

A Thesis

Submitted to the Faculty

of

Purdue University

by

Eric P. Villasenor

In Partial Fulfillment of the

Requirements for the Degree

of

Masters of Science in Electrical and Computer Engineering

December 2007

Purdue University

West Lafayette, Indiana

To Boo, who knows how to relax. To my family, thank you all for your support.

ACKNOWLEDGMENTS

This project has undergone several makeovers until finally settling down into this work. I would like to extend thanks to Mithuna Thottethodi for his direction, assistance, and ideas on this project. I would also like to thank Daeho Seo for his help and work on the project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

TCM	Traditional Clustered Microarchitecture
IBCM	Incomplete Bypass Clustered Microarchitecture
FU	Functional Unit
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
EPI	Energy Per Instruction
EX	Execute Stage
ALU	Arithmetic Logic Unit

ABSTRACT

Villasenor, Eric P. M.S.E.C.E., Purdue University, December, 2007. Intra-level Incomplete Bypassing: Achieving Performance and Power Efficiency . Major Professor: Mithuna S. Thottethodi.

Researchers have proposed clustered microarchitectures to capture the benefits of high performance and high energy efficiency. Typically, clustered microarchitectures offer fast local bypasses (i.e., value forwarding between instructions) within clusters and require global bypasses to take longer, more than one cycle. With communication locality (i.e., most communication is within the clusters) the clustered designs capture the benefits of both improved instructions per cycle and increased clock-frequency. Traditional clustered microarchitectures are implemented by partitioning the register file and associated functional units to clusters. In this work, an alternate technique is demonstrated – Incomplete bypassing – to achieve similar clustering. Incomplete bypass based clustering is similar to traditional clustering in that it creates groups of functional units where intra-group communication occurs within a single cycle over fast bypass wires and inter-group communication takes longer, more than one cycle. One key difference is that in traditional clustered microarchitectures, inter-cluster communication takes place over the global buses whereas incomplete bypass designs achieve inter-group communication via the register file. It is demonstrated that incomplete bypass based clustered micro-architecture achieves higher performance (10% speedup) and better energy efficiency than traditional clustered microarchitectures.

1. INTRODUCTION

1.1 Motivation

Even as general-purpose computing moves toward multicore/manycore designs as the mainstream model, the design of a single core to achieve high single-thread performance and power-efficiency remains a key design consideration. Much research over the past decade has focused on precisely this goal. One important example is the clustered architecture which effectively groups, or clusters, execution units such that intra-cluster communication, inside the cluster, is fast and inter-cluster communication, between clusters, is (relatively) slow [1]. Clustering helps increase clock speed because only the intra-cluster communication has to occur within a single clock cycle. Furthermore, because instructions can be steered to clusters such that intra-cluster communication is the common case and inter-cluster communication is rare, the decrease in instructions per cycle (IPC) is minimal.

While the general benefits of clustering: reduced complexity, increased performance, and improved power, are well known, the implementation of clustering can have a significant impact on the overall performance of the architecture. There are many renditions of clustered architecture implementations, the one referred to in this work is the implementation in which a cluster has a local register file and local functional units, this implementation will be referred to as the traditional clustered microarchitecture (TCM). This work describes an alternate implementation of clustering — Incomplete Bypass-based Clustered Micro-architectures (IBCM) — that achieves 10% better performance than traditional clustered microarchitectures (TCM) while simultaneously improving energy efficiency in the hottest part of the core.

In *IBCM*, like in *TCM*, instructions that are scheduled on functional units within a cluster can forward/bypass results among themselves in a single cycle (i.e., they are bypass-connected). However, unlike *TCM*, instructions executing on functional units that are not within a cluster cannot bypass results to each other at all in *IBCM*. Any communication between such instructions occurs via the register file. At first glance, it may appear that *IBCM* is replacing fast and cheap wire-based communication with expensive, register based communication. However, a careful analysis of *IBCM* reveals the following three key insights due to which it achieves higher performance and better energy efficiency.

First, it is demonstrated that, in the common case, the *bypass fanout* — the number of times a value is read from the bypass network — is typically a small number. Fewer than 2% of instructions in the integer SpecCPU 2000 benchmark suite have a *bypass fanout* greater than 2. This gives the insight that full bypass networks which connect the output of every functional unit to the two inputs of every other functional unit are over-provisioned. As such, *IBCM* truncates the result bus to limit the connectivity among functional units. The truncation of result buses reduces the length of wires, which in turn improves both the clock cycle and the energy consumption. This is achieved because driving shorter buses is faster and consumes less energy. It is shown later in Section 3.1, the truncated result buses of *IBCM* are shorter than the intra-cluster bypass buses of *TCM* and can directly lead to a faster clock. Because the truncation of the bypass wires eliminates bypass connectivity of some functional units, any communication between instructions will have to occur via the register file.

Secondly, overall performance is relatively insensitive to increases in the delay of inter-cluster communication, a fact that *IBCM* exploits. For example, increasing the inter-cluster communication delay from one cycle to two cycles reduces the IPC by less than 2%. This is not suprising because inter-cluster communication is not the common case. Thus, the same property that enables clustering – locality of intra-cluster communication – makes it tolerant of increased inter-cluster communication latency. This is a key observation because the design choice of forcing inter-cluster communication to occur via the register file increases the communication latency to two cycles. This two cycle inter-cluster latency consists of one cycle to write to the register file and another cycle to read from the register file.

Finally, the above discussion focuses purely on the "Execute" (EX) stage (including ALU delay and bypass delay). Though the EX stage, with its EX-EX pipeline loop, is known to be clock-critical, it cannot be assumed that a reduction in the delay of the EX stage leads to a corresponding increase in clock speed. This is because other stages (and the issue logic stage, in particular) may limit the reduction in clock cycle time. However, it is shown later in Section 4.3 that there are ways to sacrifice some ILP to ensure that the processor can be operated at a faster clock cycle that is limited only by the EX stage delay.

On energy efficiency, as measured by energy per instruction, *IBCM* is expected to be more energy efficient because of the shorter intra-cluster result buses that are driven in the common case. Further, *IBCM* eliminates the additional cost of copying over a global bus for inter-cluster communication. One caveat is that these energy savings occur only in the EX stage and are thus modest when considered over the entire pipeline. However, improving the energy efficiency in parts (register file and ALU output drivers) that are known to be among the hottest regions [2] in a core remains an important advantage of *IBCM*.

In summary, the two key contributions of this work are as follows. First development of *IBCM*, an implementation of clustering based on incomplete bypass networks. Compared to *TCM*, *IBCM* achieves 10% better performance by reducing the clock-critical delay of the EX stage by 13% while degrading IPC by 2%. Finally it is demonstrated that *IBCM* is more energy efficient than *TCM*. This is not surprising because in the common case, shorter result buses are driven by each instruction's output and *IBCM* completely eliminates the additional cost of inter-cluster communication. While there is little improvement when considering the pipeline as a whole, *IBCM* does improve the energy efficiency in the hottest parts of the chip.

1.2 Overview

The rest of this thesis is organized as follows. Chapter 2 discusses the background of clustered architectures as well as the opportunity for incomplete bypass based clustering. Chapter 3 describes the implementation of *IBCM* and the scheduling techniques used for the *IBCM* design. Discussion of the simulation model, evaluation methodology, and results are presented in Chapter 4. Chapter 5 discusses related works, summarizes, and concludes this thesis.

2. BACKGROUND AND OPPORTUNITY STUDY

This chapter characterizes the *bypass fanout* of instructions which serves to illustrate the opportunity for incomplete bypass based clustering designs. Finally, a brief overview of the functionality of bypass networks and traditional clustering implementations is presented.

2.1 Characterizing Bypass Utilization

In this section, utilization of bypass networks is characterized. The insights from this evaluation directly leads to the conclusion that fully-connected bypass networks are over-provisioned and that there is abundant scope for realizing comparable performance with incomplete bypass networks. Extraneous communication exist in these over-provisioned bypass networks that incomplete bypass networks are able to alleviate. It also offers insights to the type of incomplete bypass networks that can exploit the observed patterns of bypass network utilization.

Bypass networks are an integral part of high-performance processor design. They enable improved performance by alleviating the data hazards inherent to program execution. Thus, bypass networks enable back-to-back issue of dependent instructions. Bypass networks are also known to be a source of delay complexity in processor designs [1, 3]. The network complexity is a function of issue width and grows quadratically as it increases. This is because every output of a functional unit is connected to the inputs of every functional unit.

2.1.1 Bypass Fanout Evaluation Methodology

Simplescalar 3.0 [4] is used to simulate two processor configurations shown in Table 2.1. One of the machines is a 4-way issue processor with realistic memory hierarchy, branch predictor and single-cycle issue logic. Note, single-cycle issue logic is a conservative assumption since it potentially exposes more bypass utilization. An aggressive configuration is also simulated with aggressive resources, perfect memory, perfect branch prediction, and single-cycle issue logic. This aggressive configuration serves to expose more ILP (and potentially more bypass utilization) as it eliminates the sources of pipeline stalls. The integer SpecCPU 2000 benchmark suite is simulated with reference data sets for 100 million instructions after fast-forwarding to the most representative simulation point indicated by the SimPoint 3.0 Tool-set [5]. Profile information is represented in Figure 2.1 and Figure 2.2, for both machine configurations in Table 2.1.

2.1.2 Bypass Fanout Metric

The metric used to measure utilization of a bypass network is "*bypass fanout*." The *bypass fanout*, of an instruction i, is defined as the number of times the value produced by the instruction i is read from the bypass network before the value is written to the register file. Butts and Sohi describe another closely related concept —*degree-of-use*— which counts all consuming instructions including those that read the produced value from the register file [6]. However, utilization of the bypass network remains the only item of interest for this concept of *bypass fanout*.

Isolating the *bypass fanout* from the overall *degree-of-use* offers an interesting challenge. Unlike *degree-of-use*, which is unique to a given control path, *bypass fanout* is sensitive to instruction scheduling as well. If the consuming instructions are scheduled soon after the producing instruction, the consumed value is likely to be sourced from the bypass network. On the other hand, if the consuming instructions are delayed for any reason, the consumed value may be read from the register file. The

Table 2.1
Bypass fanout machine configuration.

Parameter	**Real**	**Aggressive**
Issue Width	4	8
Commit Width	4	8
Branch Prediction	14 bit g-share	Perfect
Physical Registers	256	256
Load Store Queue	128	128
L1 Cache	32KB(I)+32KB(D) Direct Mapped	Perfect, Infinite
L1 Latency	1 cycle	–
L2 Cache	2MB Unified 4-way, 64B	–
Mem. Latency	200	–

reasons for temporal separation between producer and consumer instructions could be one of the following. Temporal separation would be intrinsic to the application (e.g. distance in the data-flow-graph), extrinsic artifacts of the processor such as resource constraints (e.g. issue width), pipeline inefficiencies (e.g. lack of back-to-back dependent instruction issue), intervening branch miss predictions, cache misses, or exceptions. Alternatively, cache misses may also delay value production. The delay in value production may bring consumers closer together to increase the *bypass fanout* of certain instructions.

Bypass fanout is measured by simulation on both a realistic and an aggressive configuration (Table 2.1) to estimate the variation introduced by the factors mentioned above. Fortunately, these experiments indicate that one key trend is insensitive to such factors. The trend is that the *bypass fanout* of instructions is less than or equal to 2 in the common case.

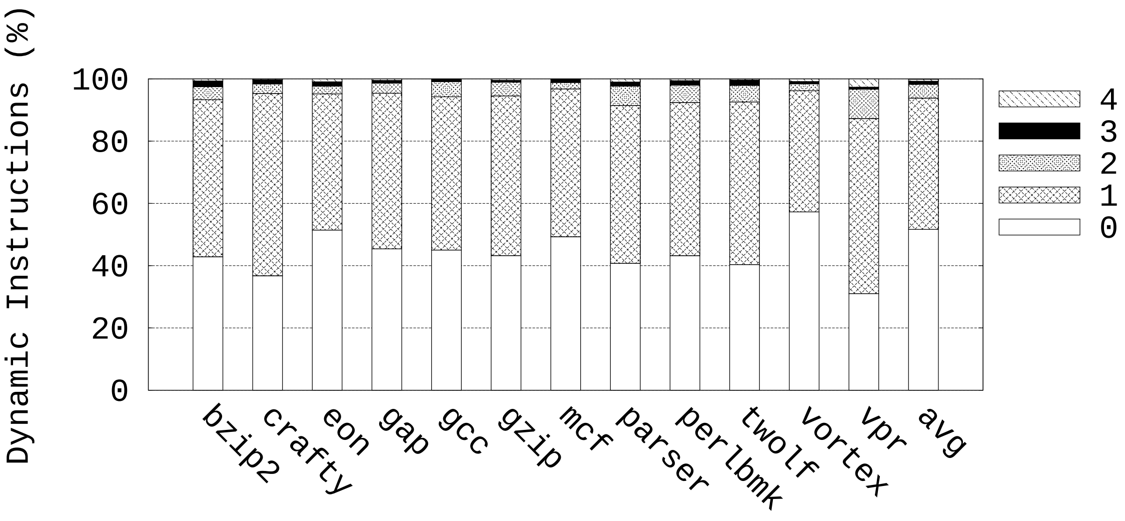

Fig. 2.1. Realistic bypass fanout profile.

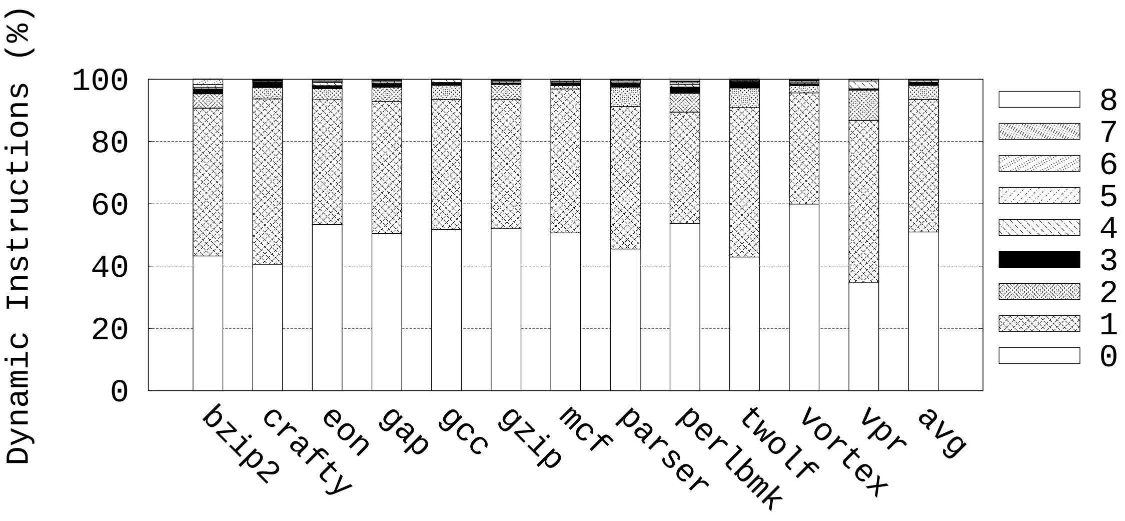

Fig. 2.2. Aggressive bypass fanout profile.

Profile data for the bypass fanout of the integer SpecCPU 2000 benchmark suite is presented in two graphs; Figure 2.1 represents the realistic configuration and Figure 2.2 represents the aggressive configuration. Table 2.1, again, depicts the configurations for these two machines. These figures plot the percentages of dynamic instructions with various possible *bypass fanout* values for the two machine configurations. Each graph (Figure 2.1 and Figure 2.2) contains twelve bars corresponding to the SpecCPU 2000 integer benchmark suite; in addition, the rightmost (thirteenth) bar displays the average across all twelve integer benchmarks.

The primary observations from Figure 2.1 and Figure 2.2 are as follows. First, on average, very few instructions (1.8%) have a *bypass fanout* greater than 2 on the 4-way superscalar configuration (Figure 2.1) with `vpr` reporting the maximum of 3.3%. Second, on the aggressive configuration (Figure 2.2), the fraction of instructions reporting a *bypass fanout* greater than 2 increases marginally to 2%. `bzip2` is the benchmark with the largest fraction at 4.7%. Finally, the fact that the bypass fanout profiles, under both the aggressive and realistic configurations, are heavily skewed towards low ($\leq$2) fanout values indicates that low-bypass fanout is a fundamental program property.

These *bypass fanout* measurements reveal significant underutilization of the bypass network with more than 98% of instructions having a *bypass fanout* of 2 or less when averaged across all integer benchmarks of the SpecCPU 2000 benchmark suite. This underutilization clearly implies that bypass networks are over-provisioned for the use they typically endure. Exploiting this common case is the key idea behind types of incomplete bypass-based networks.

2.2 Clustering Background

In this section, the background of bypass networks and how clustering operates will be discussed. Then, the insights of low *bypass fanout* and bypass network implementation are combined to describe the *IBCM* design.

In all, bypass networks can become a major source of delay and complexity in the execution unit of a processor. Clustering as stated, reduces the complexity of these networks by grouping functional units into clusters. This decrease in the complexity and delay of such networks is a boon that clustered architectures exploit. However, traditional clustered architectures only reach so far before diminishing returns are encountered.

Incomplete bypass networks offer an interesting solution in achieving performance and energy improvements beyond those offered by traditional clustered architectures. Because this technique of incomplete bypassing relies on clustering via incomplete bypass networks, background information is provided on the operation of full bypass networks in the basic monolithic superscalar architectures (Section 2.2.1) and the operation of *TCM* (Section 2.2.2).

2.2.1 Monolithic, Full Bypass Networks

The schematic diagram in Figure 2.3 illustrates the layout of a fully connected EX-EX bypass network as described in [3]. This layout has also been used by Brooks *et.al.* [7] and is representative of the layouts of the MIPS R10000 [8] and the Alpha 21264 [9] processors. The centrally located register file offers two dedicated read ports and one dedicated write port for each functional unit. The outputs of the functional units are placed on result buses that span the width (due to reoriented diagrams, "width" is used wherever [3] uses "height") of four functional units and the register file. The same widths are used for functional units and register files as listed in [3] and assume identical general-purpose integer ALUs. Discussion is restricted to integer programs and integer functional units. The general principals extend to floating point functional units as well, though the parameters will be different. The results placed on the result buses are available at the operand multiplexers (shown as shaded rectangles) associated with the inputs of each functional unit. The operand multiplexer may place data from the result bus (instead of the data that has been

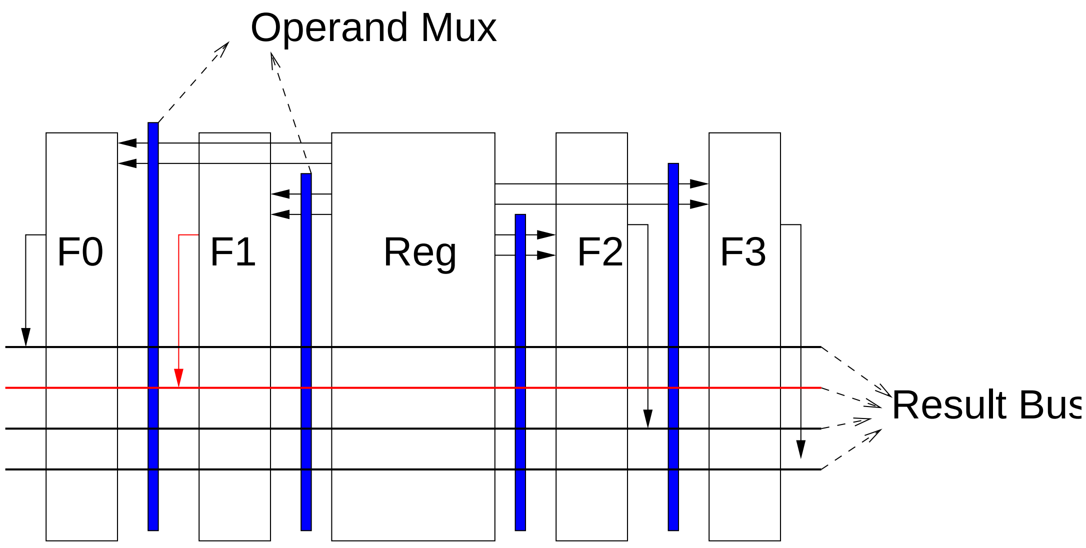

Fig. 2.3. Layout of 4-wide fully-connected, single-cycle bypass network.

read from the register file) at the input of the functional unit. The reader is referred to [3] for additional implementation details.

Three observations can be made from the layout of the fully connected bypass network. First, the only consequence of eliminating the bypass path to one of the inputs of a functional unit is the removal of one of the two connections from the result bus to the operand multiplexer. The length of the result bus is unchanged because the value still has to be supplied to one of the inputs. The result bus is the dominant contributor to overall bypass wire delay because the capacitance of the result bus dominates the total capacitance observed by the ALU output drivers. Ahuja *et.al.* [10] propose one optimization: to eliminate the bypass path to one of the inputs and leverage commutativity of the operation to overcome the resulting pipeline stalls. From the above observation, it follows that the presented optimization has very little benefit for delay complexity but has the potential pitfall of introducing interlock delays. Second, if one ALU output is not bypassed at all, it results in the complete elimination of one of the result buses. This may help reduce wire density, but the overall bypass delay is still bounded by the delay of the other result buses which are unchanged in length. Finally, the above two observations rule out two types of incomplete bypass networks. The only remaining option is to limit the horizontal extension of the result buses. This results in the output of functional units being bypassed to some functional units but not to others. Furthermore, the reduction in the horizontal reach of the result buses must occur for every single result bus since the delay complexity depends on the longest result bus.

Incomplete bypass network designs can be derived naturally from the above three observations. This insight will aid in designing incomplete bypass based clustered microarchitectures in Section 3.1, after discussing the operation of traditional clustered microarchitectures.

2.2.2 Traditional Clustered Micro-architectures

Though there are many flavors of the clustered architecture implementations, this thesis refers to the implementation in which a cluster has a local register file and local functional units, as the traditional clustered microarchitecture (TCM). Figure 2.4 illustrates the layout of the register files, functional units and bypass network in a 2-cluster *TCM*. Each cluster has a register file and half as many functional units as the monolithic organization. The intra-cluster result buses are shorter in length by the width of two ALUs. It is the delay of driving this shorter bus (in addition to ALU delay and clock overhead) that must be accommodated in a single clock cycle. Inter-cluster communication on the global bus (shown with dotted lines) takes two cycles. Note, each cluster-local register file has half as many read ports as the register file in the monolithic architecture. Every instruction that creates a new value is propagated to both the clusters. This is a conservative assumption for performance because it minimizes the number of pipeline stalls. However, replicating every value is not ideal from an energy-efficiency perspective. When energy efficiency is discussed, this assumption that every value is replicated in both register files will be relaxed.

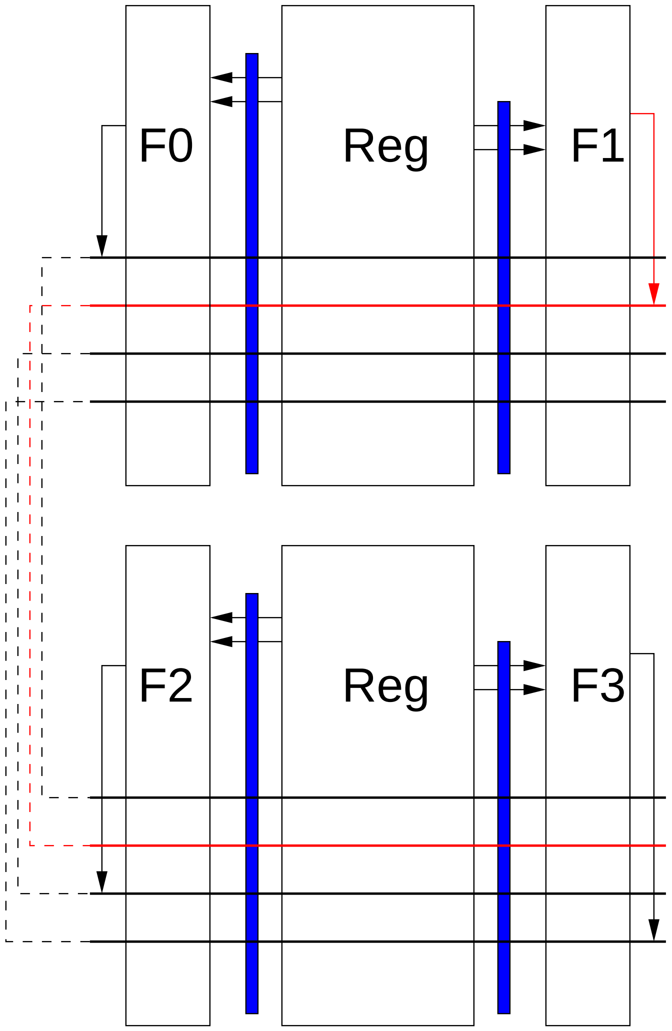

Fig. 2.4. Layout of 4-wide, traditional clustered microarchitecture.

3. INCOMPLETE BYPASS CLUSTERED MICROARCHITECTURE

This chapter introduces *IBCM* and the insights that make it possible. Finally, scheduling techniques of clustered architectures are discussed.

3.1 *IBCM* Layout

Consider the spectrum of designs that vary in their bypass connectivity. At one end of this spectrum, there are the traditional fully-connected bypass networks (Figure 2.3). At the other end of the spectrum, there are architectures with no bypass/forwarding where all value communication occurs via the register file(s) and/or memory. *IBCM* is an intermediate design point in this spectrum in which the bypass network spans exactly two functional units (in a 4-wide superscalar machine) as illustrated in Figure 3.1.

IBCM exploits the previously mentioned intuition that the incomplete bypass networks of interest are the ones which limit the horizontal reach of the result buses. A key differentiating factor in this layout (compared to the layout of *TCM*) is that the result buses do not have to span the register file. This results in a disproportionate reduction in wire delay (35.7%, as estimated using ITRS roadmap parameters for 50nm technology [11]) since even a banked register file is significantly wider than the ALUs. A simple observation, yet it provides astounding opportunity for design improvement. It will be shown later, in Section 4.2 that the reduction in wire delay causes an overall reduction of 13% in the EX stage delay (including ALU delay and latch overheads).

From the layouts, it can be seen that the implementation of *TCM* and *IBCM* designs are clearly different. For example, *TCM* architectures, unlike *IBCM* archi-

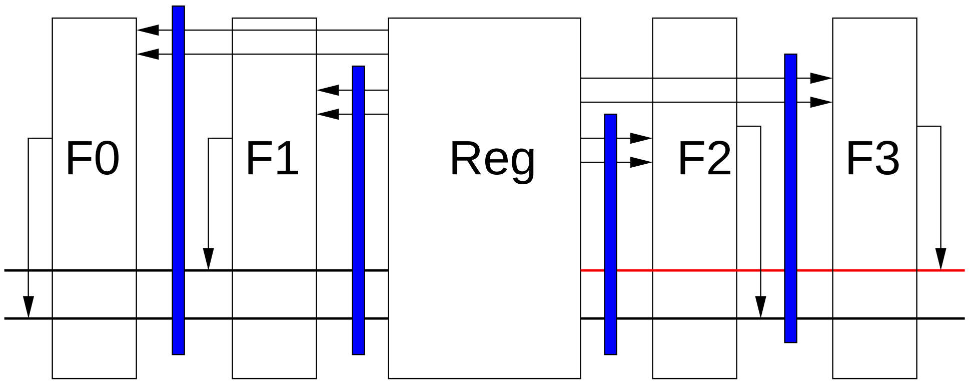

Fig. 3.1. Incomplete Bypass-based Cluster Micro-architecture.

tectures, use replicated register files and hierarchical bypass networks. Yet, the underlying clustering approach remains similar as both *TCM* and *IBCM* architectures deliver non-uniform latencies for inter-instruction value communication. Consumer instructions which issue to the same cluster as the producer ALU enjoy fast bypass in *TCM* as do consumer instructions that issue to a bypass-connected ALU (i.e., an ALU on the same side of the register file) of the producer ALU in *IBCM*. Consumer instructions which issue to a different cluster in *TCM* architectures suffer a longer latency penalty since operands must be communicated over the longer inter-cluster bypass wires. Similarly, consumer instructions which issue to an ALU not on the same side of the register file as the producer ALU suffer the delay of communication via the register file.

In summary, presented herein is an alternate implementation of clustering – *IBCM*– based on incomplete bypassing. It can be seen that *IBCM* provides a level of clustering without the overhead of *TCM* components. This feature is a boon to *IBCM* as it shifts the point of diminishing returns for clustered architectures.

3.2 Cluster Scheduling

The monolithic processor organization with uniform all-to-all bypass connectivity provides one major benefit – any ready instruction may be assigned to any free functional unit. As such is the benefit of single issue windows, partitioned instruction issue windows are assumed for *TCM*. Cluster implementations, in contrast to single window schedulers, rely on appropriate scheduling of instructions to functional units to avoid/minimize inter-cluster delays. The fact that *IBCM* is an alternative implementation of clustering renders the entire body of literature that deals with scheduling for *TCM* architectures directly applicable for this design as well [1, 12–17]. These partitioned issue windows have previously been proposed for *TCM*s [1]. This design (*TCM*) associates an instruction window with each cluster. Instructions are steered at instruction dispatch to one of the cluster issue windows. Once steered, instructions remain in that cluster until they execute. Any communicated values are shared via the result buses depicted in Figure 2.4, scheduling policies are used to minimize the need for inter-cluster communication. A representative (but not exhaustive) set of instruction steering policies include: dependency-based steering [1, 12], criticality-heuristic-based steering [13–15] and instruction replication [16, 17] policies.

3.2.1 Single Window Schedulers

Traditional schedulers consist of wakeup and select stage logic. Wakeup and select are, in general, thought of as one atomic operation, this is shown to be complex and infeasible for wide issue configurations [1, 15]. This insight leads to the determination that, for single issue window schedulers, they become clock-critical as a pipeline stage. Figure 3.2 depicts this traditional scheduler. Figure 3.3 illustrates traditional wakeup logic for one source operand [18].

Traditional wakeup logic functions as follows. First, it determines when an instruction is awoken by waiting for the determined latency after their source operands are broadcast. After the source operands are ready, and the operational latency has expired, the instruction is ready to issue. The select stage, also depicted in Figure 3.2, then seeks a suitable functional unit and issues the instruction.

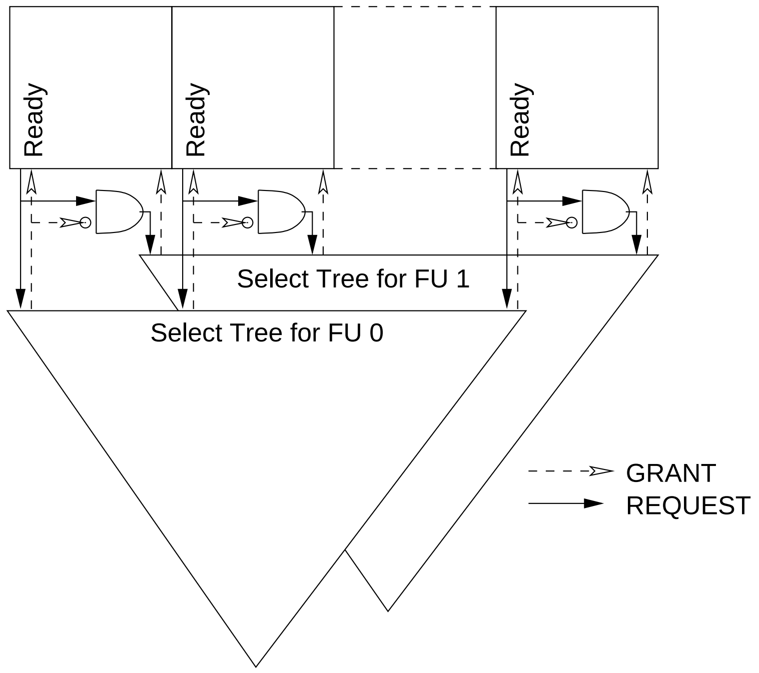

Fig. 3.2. Single window scheduler.

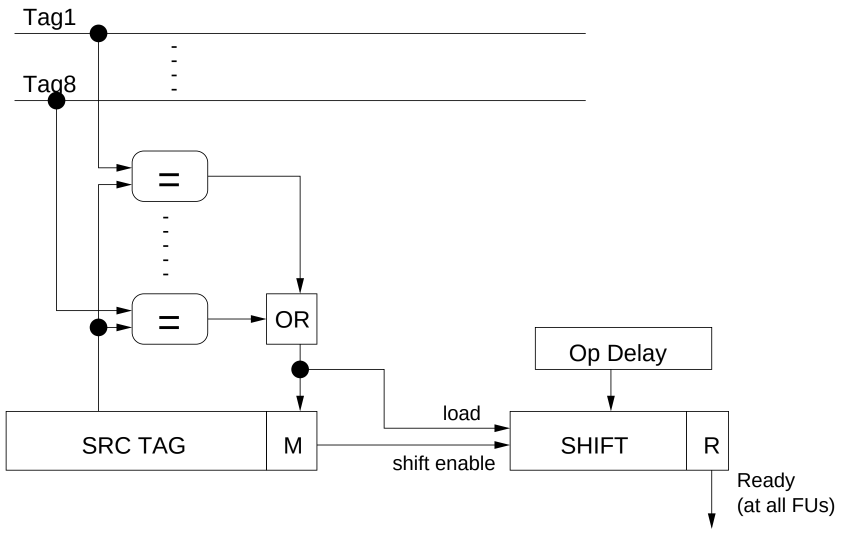

Fig. 3.3. Traditional wakeup logic.

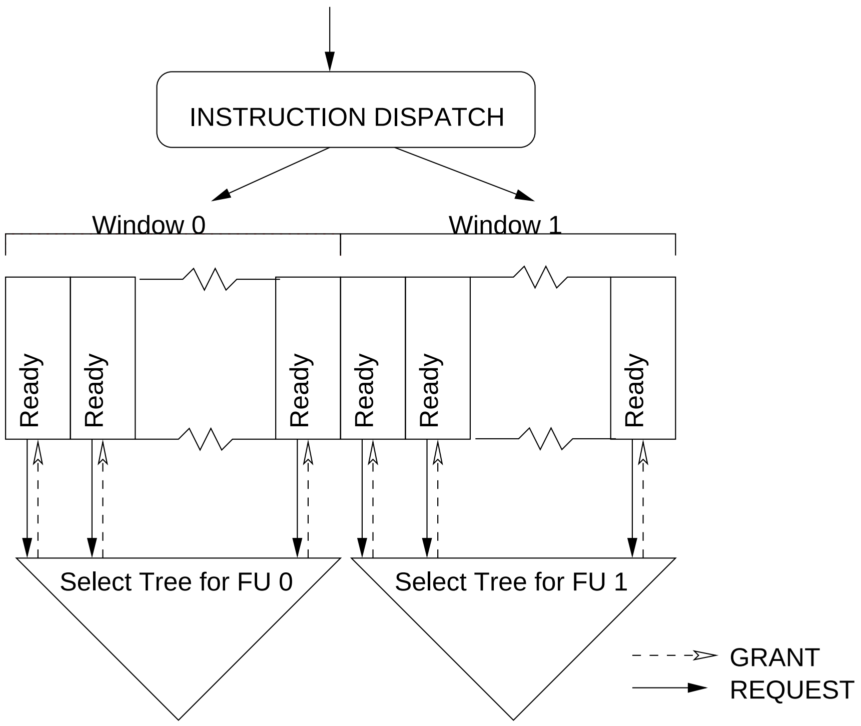

Fig. 3.4. Partitioned window scheduler.

3.2.2 Partitioned Window Schedulers

Partitioned window schedulers significantly reduce complexity for wakeup logic. A single window scheduler contains wakeup and select logic; however, by sectioning off functional units, the complexity of this logic is reduced. Thus, partitioned window schedulers reduce the schedulers clock-critical status in the processor.

A partitioned scheduler is illustrated in Figure 3.4. Again the wakeup logic is the same as in Figure 3.3. It requires that the wakeup logic determine if the instruction is ready at it's cluster window (it may be surmised in Figure 3.4 that each cluster contains one functional unit), this reduces the number of available functional units which the select logic views. In order to benefit from the reduced complexity, intelligent steering policies must be established to ensure an efficient flow of instructions, as mentioned above.

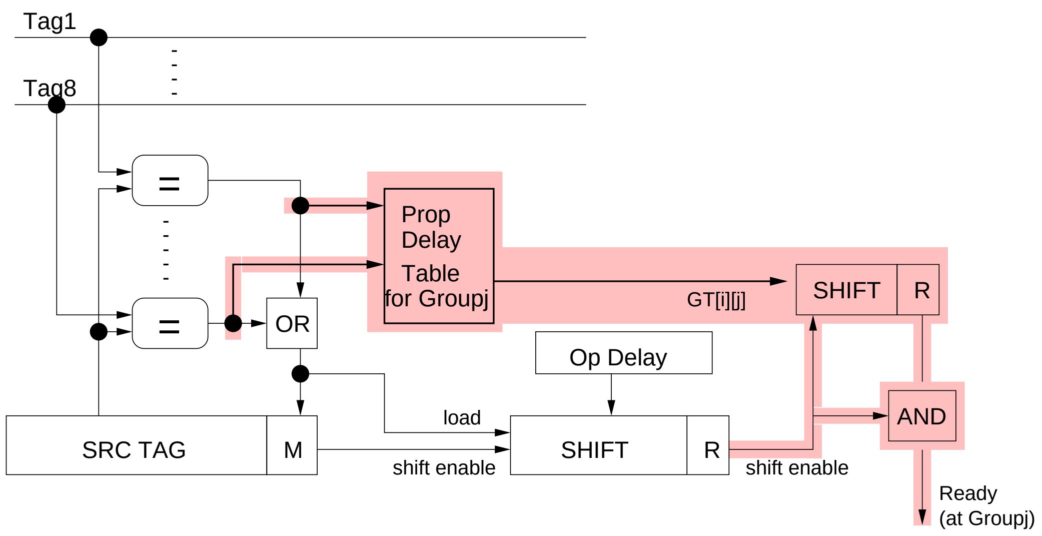

Fig. 3.5. *IBCM* partitioned window wakeup logic.

3.2.3 *IBCM* Scheduler Modifications

Slight modifications to the partitioned window scheduler must be made in order to allow incomplete bypass-based designs to maintain efficient instruction flow. Note, it was stated before in *TCM* a cluster was associated with a window; however, bypass-connected ALUs, referred to in this section as groups for brevity, will now be associated with a window. One will recall, from Section 3.1, that communication between groups suffer a delay through the register file.

The delay between groups can be represented by a group table, or a propagation delay table. A processor with G groups would have a table with G entries. This table contains latency from every other group to the group associated with this window. The table inherently maintains the property that number of groups is not linearly related to issue width.

The modifications made to traditional wakeup logic, the shaded regions in Figure 3.5, are to account for the group delays described above. Figure 3.5 focuses on the logic for a single operand. The logic functions as follows. First, the tag line which matches the source tag is the group where the dependency is located. Second, the logic then looks into the propagation delay table for the latency between groups. Third, the propagation latency is loaded into a shift register. Fourth, the delays are then counted down, until the Ready flag is asserted. Finally, the instruction is ready, select logic finds an available functional unit in the group and the instruction executes.

3.2.4 *IBCM* Steering Policy

As a proof-of-concept, the following adaptation of the dependency-based instruction steering policy is evaluated for *IBCM*. The partitioned window scheduler as described in Section 3.2.3 allow for incomplete bypass-based policies. Where *IBCM* has need of incomplete-bypass awareness, the *TCM* does not, and therefore issue windows remain associated with a cluster, rather than a group. Each instruction is steered to the window of the group where its source operand(s) is(are) produced, by indicating an instruction is steered to a group, it is dispatched to an issue queue window associated with that group. Issue queues are load balanced to prevent any dependence chains from overloading any subset of the processors resources.

Several scenarios exist for scheduling instructions with this policy. An instruction with no dependencies is free to move to any window, or queue, as long as there is room. With load balancing, the window with the least pressure becomes the initial target of the scheduler. This assures an even distribution of independent instructions, which may head a dependency chain. An instruction with one dependency targets the window of the source, or parent, instruction. If the source window is full, the instruction is steered to another issue window irrespective of the register dependence, again with load balancing the target then becomes the window with the least pressure. An instruction with two dependencies targets the window of its parents. This is trivial if both parents are on the same group, the instruction will target that window. If the window is full, it will again go to the window with the least pressure. With parents on separate groups, the instruction targets a window associated with a parent instruction.

Should one of the parent windows have less pressure than the other, the instruction targets that window. Should either be full, the instruction will target the one that is not full. Should neither source window present a valid target, the window with the least pressure is targeted, irrespective of the register dependencies.

TCM operates in a similar fashion to *IBCM* but at the cluster level rather than the group. Each instruction is steered to the cluster of the functional unit where its source operand(s) is(are) produced, by indicating an instruction is steered to a cluster, it is dispatched to an issue queue window associated with that cluster. If the source instructions are in different clusters, the instruction may be steered to the cluster with the least number of queued instructions. Finally, if one of the issue windows is full, instructions are steered to the other issue window irrespective of the register dependencies.

4. RESULTS

First, this chapter elaborates evaluation methodology, simulation models, and architectural enhancements to the simulation model. Second, contained herein is discussion which allows the execute stage to attain clock-critical status. Third, it presents clock-critical latencies in the execution unit. Finally, simulation results are discussed.

4.1 Evaluation Methodology

Though some of the following text may appear repetitive with the methodology used to measure *bypass fanout*, there are significant differences since the purpose of the two sets of simulations are quite different.

SimpleScalar 3.0 [4] is used to model the architectural modifications and measure their impact on a superscalar processor using the Alpha ISA. The configuration of the simulated processor is depicted in Table 4.1. Two slightly different configurations, realistic and ideal, are used in the simulations. The realistic configuration has no load speculation and has a combining branch predictor. The ideal configuration is used to rule out the possibility that branch prediction or memory dependencies are choking ILP (and resulting in a weakened base configuration), the ideal configuration has perfect branch prediction and perfect load dependence prediction. With the perfect load dependence predictor, a load is issued as soon as the last store to the same address is ready. A store is defined as ready if its effective address has been calculated and its operands (from which the value to be stored is calculated) are ready. It is assumed that a load directly reads its value from the last store using the load/store queue. Integer benchmarks of the SPEC CPU 2000 benchmark suite are used with reference data sets for 100 million instructions [19]. The benchmarks are fast forwarded to the most representative section using Simpoints 3.0 tool set and the early simulation points for each respective benchmark [5]. Wattch [7] is used to measure energy usage and power consumption of the processor. Single cycle issue logic is assumed to expose more bypass network utilization. Also the register file is replaced by a banked register file design [20]. This banked register file is implemented with two read ports, and two write ports, as described in [20] with slight modifications. Bypasses from different clusters are handled as bank conflicts, also the read ports are fully networked to the functional units. Base functional unit latency is not modified, only the incurred latency is added to the base.

Though discussion is limited to integer ALUs, bypassing from/to the two load ports is included. For both the *TCM* and *IBCM* configurations, it is assumed that each load port is fully connected by bypasses, and equidistant from all functional units with a latency of 2 cycles. The integer multiplier is not included in the bypass model as there were very few ($\leq$0.5%) multiply instructions in the integer SPEC2000 benchmarks. The simulations model partitioned issue queues with a dependency-based steering policy, as mentioned earlier in Section 3.2.4. There have been other studies that examined more sophisticated instruction steering heuristics. Because such steering policies work by minimizing the number of inter-cluster communications, they will benefit both *TCM* and *IBCM* and are not likely to alter the results of the simulations.

4.2 Clock Critical Latency

ITRS roadmap numbers are used to compute the delays of wires in the 50nm technology with intermediate metal layer wires [11]. Being conservative, it is assumed that bypass wires use repeaters that are placed at register-file/ALU boundaries. This use of repeaters reduces the delay of the bypass wires in the base-case. Since the wire delays depend on lengths of wires between repeaters, register-file and ALU dimensions obtained from the literature are scaled to 50nm technology. It is further assumed that the EX stage delay is composed of three components: ALU delay (190 ps, obtained

Table 4.1
Simulation processor configuration.

Processor Configuration	
Fetch, Decode, Issue, Commit Width	4
ROB Entries	128
Issue Queue	2 window x 32 entries each (Except *IBCM*-IQ/2)
Load Speculation	Realistic: No load speculation Ideal: Perfect load dependence prediction
Branch Prediction	Realistic: Bimodal & 2-level predictor combined, BTB 2048, RAS 64 Ideal: Perfect prediction
Memory Hierarchy	
L1 Instruction and Data Cache	2-cycle, 32Kb, 4-way, 32-byte blocks, LRU
L2 Unified Cache	8-cycle, 1MB, 8-way, 64-byte blocks, LRU
Memory Latency	400 cycles
Memory Bus Width	16 bytes
Instruction and Data TLB	64 sets, 4-way, LRU

from scaling previously published ALU delays [21]), bypass wire delay (as estimated using the method described above) and clock/latch overhead (33 ps which is approximately 10% of overall clock period in the *TCM* configuration). The ALU delay and clock overhead remain constant across all configurations. The bypass wire delay (total clock period) for *TCM* is 112 ps (335 ps) and for *IBCM* is 72 ps (295 ps). The clock frequency corresponding to the above delays are shown in Figure 4.1.

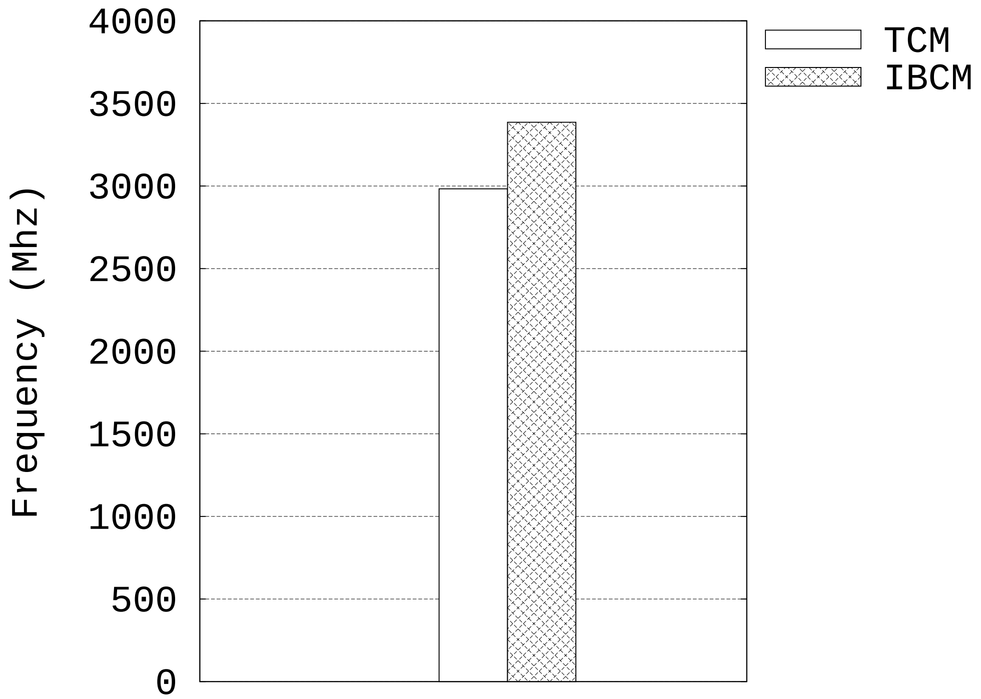

Fig. 4.1. Clock speed.

4.3 Making Execute Clock Critical

The above analysis assumes that the EX stage alone determines the clock period and all other stages are significantly sub-critical that they can be accomplished in the reduced clock cycle. This is not always true, especially since it is known that issue logic is also critical in superscalar processors.

In order to address this issue, of clock-critical stages, the following observations are made. First, it is observed that issue logic and the EX stage are singleton pipeline loops and thus cannot be pipelined without significant loss of performance. By virtue of being singleton loops, these stages will fail to benefit from back-to-back issue of dependent instructions if their logic is pipelined. The “Rename” pipeline stage is also a singleton loop. However, it is omitted from consideration because previous studies show that is is not clock-critical [1]. Second, all other pipeline stages can be pipelined without significant penalty since they do not affect back-to-back issue of dependent instructions.

From these two observations, a new configuration can be derived which has half as many issue queue entries(*IBCM-IQ/2*). It is worth mentioning that the base issue queue size for *TCM* was appropriately set (by scaling issue queue delays reported in [1] to 50nm technology) to occupy approximately 335 ps which is the clock period for the EX stage as well. By handicapping *IBCM* with an issue queue that is half the size of *TCM*, *IBCM* sacrifices some ILP. However, the issue stage is now no longer clock-critical and it can safely be assumed that the processor operates at the clock speed determined by the EX stage. The reduction in issue queue size is explicitly modeled. Deeper pipelining of other stages, in the processor, are not modeled as their impact on performance is expected to be minimal.

4.4 Results

There are two primary conclusions from the simulations. First, that *IBCM-IQ/2* achieves 10% performance improvement on average over the *TCM* configuration. The

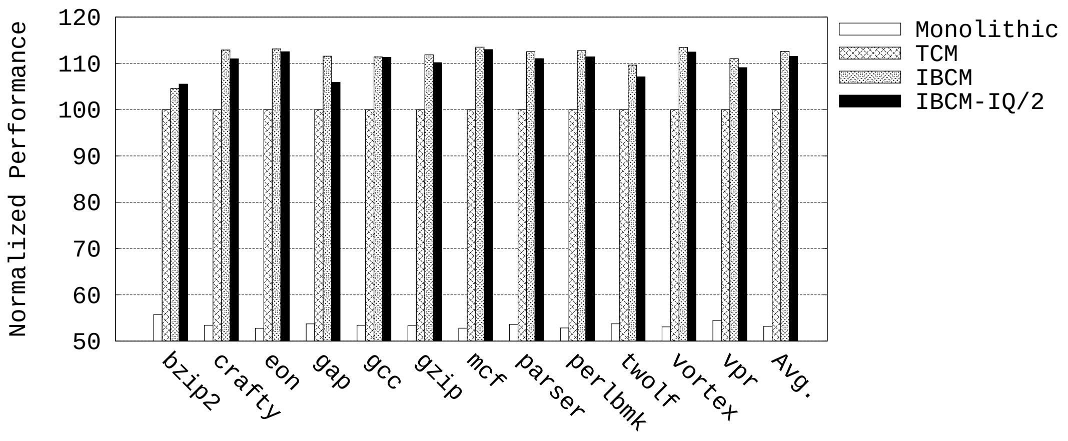

Fig. 4.2. Realistic configuration performance.

13% improvement in clock speed more than compensates for the 2% reduction in IPC due to the larger penalty for inter-cluster communication in *IBCM* and the smaller issue queue. Second, *IBCM* is more energy efficient than *TCM*, especially in the regions affected by *IBCM*. The overall reduction in *energy per instruction* (EPI) is modest (under 1%). However, when considering the register file and ALU output drivers alone, the reduction is significant, even when compared to a *TCM* with perfect future knowledge.

This section is broken into the following sub-sections. Section 4.4.1 presents the overall performance of *IBCM*. Section 4.4.2 presents the EPI measurements to support *IBCM*'s claim of energy-efficiency.

4.4.1 Performance Comparison

There are two sets of graphs for performance, Figure 4.2 for the realistic configuration and Figure 4.3 for the ideal configuration. Each graph plots the integer benchmarks from the SpecCPU 2000 benchmark suite on the X-axis (with one final additional bar showing the average across all benchmarks) and the normalized performance relative to the *TCM* configuration on the Y-axis. Each benchmark has four

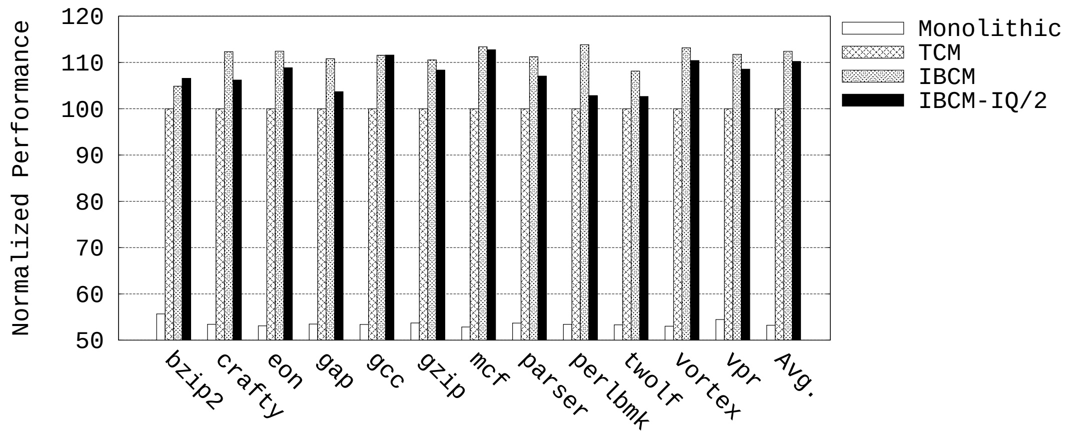

Fig. 4.3. Ideal configuration performance.

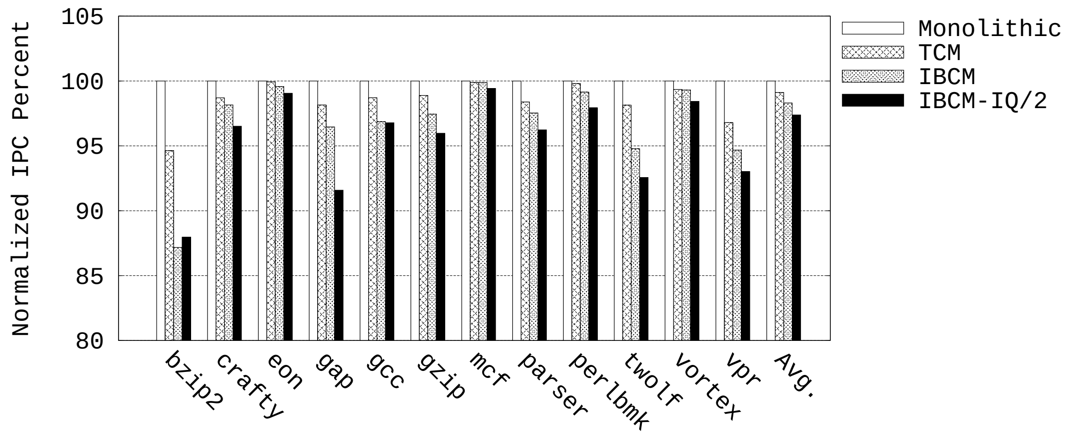

Fig. 4.4. Real configuration IPC.

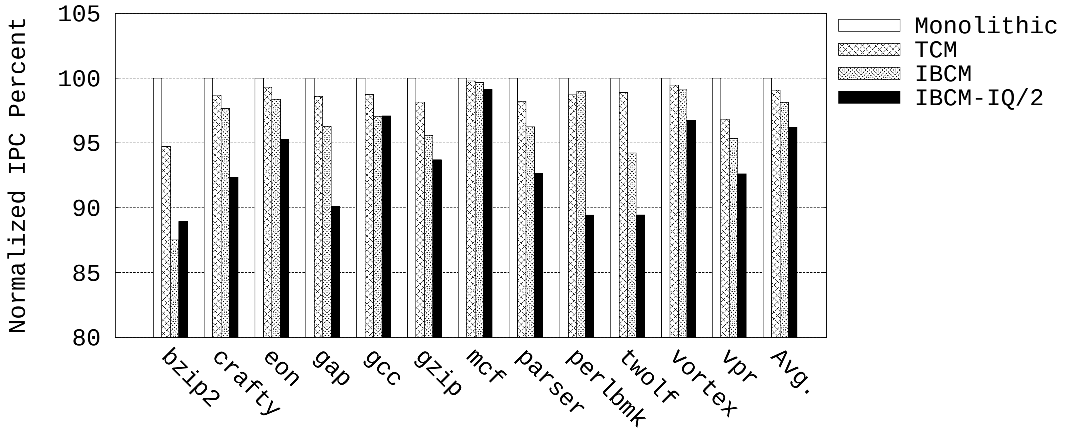

Fig. 4.5. Ideal configuration IPC.

bars corresponding to the monolithic, *TCM*, *IBCM* and *IBCM-IQ/2* configurations respectively. The monolithic bars are presented purely for completeness. The observation of note in these graphs is that the *IBCM-IQ/2* configuration out performs the *TCM* configuration by 11%.

In order to elucidate the source of these performance improvements two graphs are presented showing the IPC of the two configurations. Figure 4.4 is the IPC of the realistic configuration and Figure 4.5 is the IPC of the ideal configuration. The graph format is identical to the formats in Figure 4.2 and Figure 4.3, with the exception that the Y-axis denotes normalized IPC rather than normalized performance. As is expected, by reducing the bypasses, it is observed that going from *TCM* to *IBCM* results in a modest IPC degradation due to inter-cluster communication latency. *IBCM-IQ/2* suffers an additional 1% IPC degradation when the issue queues are reduced by half. However, the increase in frequency (13% as shown in Figure 4.1) more than compensates for the IPC loss resulting in overall performance improvement, as seen earlier. An occurrence of note, is that `bzip2` achieves better performance with a smaller issue-queue. This artifact is attributed to the way load balancing is done in the *IBCM* steering logic. While dependency based steering is the norm whenever issue queues have free slots this heuristic, the dependency rule, is violated when an issue queue is full. That is, an instruction may be issued to one cluster even though its source operands are computed on another cluster simply because the issue queue of the source cluster is full. In the case of bzip2 the 32 issue queue, which is really two partitioned issue windows of 16 entries each, resorts to such load balancing earlier than the 64 entry issue queue. This improved load balancing causes improvement in performance.

4.4.2 Energy Efficiency

The previous sections offered evidence that *IBCM* is an attractive design point from the performance point-of-view. This section evaluates the energy efficiency of

IBCM. Energy-per-instruction (EPI) is used as the metric of energy efficiency as it is independent of performance and represents the energy expended per unit of work. The power for both *TCM* and *IBCM* designs is modeled using Wattch 1.02 [7].

The baseline Wattch models are modified to include banked register files [20] and the altered result bus lengths (for both traditional and *IBCM* clustered designs). It is of note that, *TCM* requires two register files with fewer read ports in each file. Recall that the modeled *TCM* configuration assumes that writes are always eagerly propagated to the other clusters. Because this configuration may unnecessarily expend energy communicating values that will never be used, this configuration is referred to as the "brute force" (TCM BF) configuration. As one might imagine, such a configuration is a poorly conceived competitor as it inflates the EPI for the base case. This bias is eliminated by evaluating another configuration, which assumes perfect future knowledge (TCM PK) and communicates values across clusters only if those values are needed on the other cluster. This configuration, though infeasible, sacrifices no performance relative to the "brute force" clustering implementation. It is further assumed that the two clusters are adjacent to each other when calculating the length of the inter-cluster bypass bus. This assumption favors *TCM* since the clusters are actually well separated in the Alpha 21264 layout [9] which would result in longer inter-cluster buses for the *TCM* configuration.

The summarized EPI is plotted in two graphs, Figure 4.6 for the realistic configuration and Figure 4.7 for the ideal processor configuration. Each graph contains two sets of four bars. One set, which is labeled "affected", plots the EPI in the regions affected by optimizations employed in the *IBCM* design (i.e, register files and bypass network result bus drivers). The second set of bars, labeled "total", plots the EPI for the entire processor. Within each set the four bars correspond to different configurations: the perfect-knowledge (TCM PK) configuration of *TCM*, the brute-force (TCM BF) configuration of *TCM*, *IBCM* configuration, and *IBCM-IQ/2* configuration (issue queue is reduced by half).

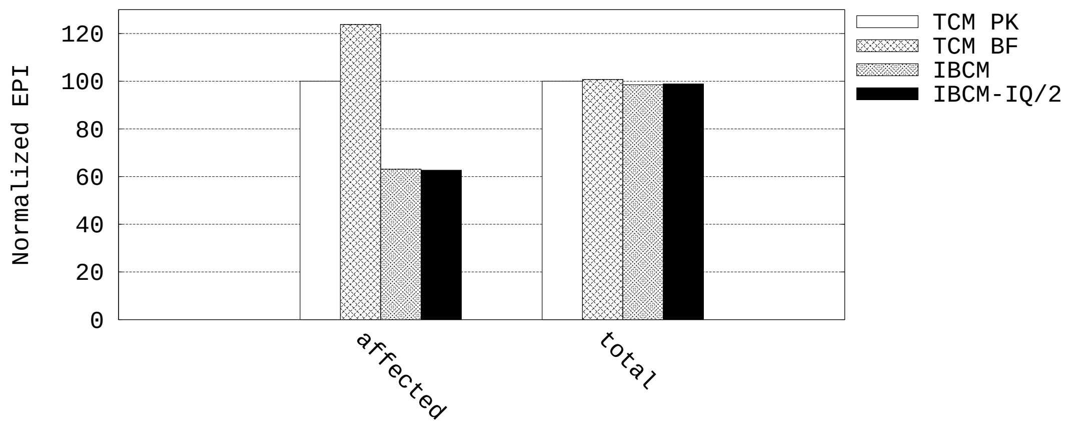

Fig. 4.6. Real configuration energy per instruction.

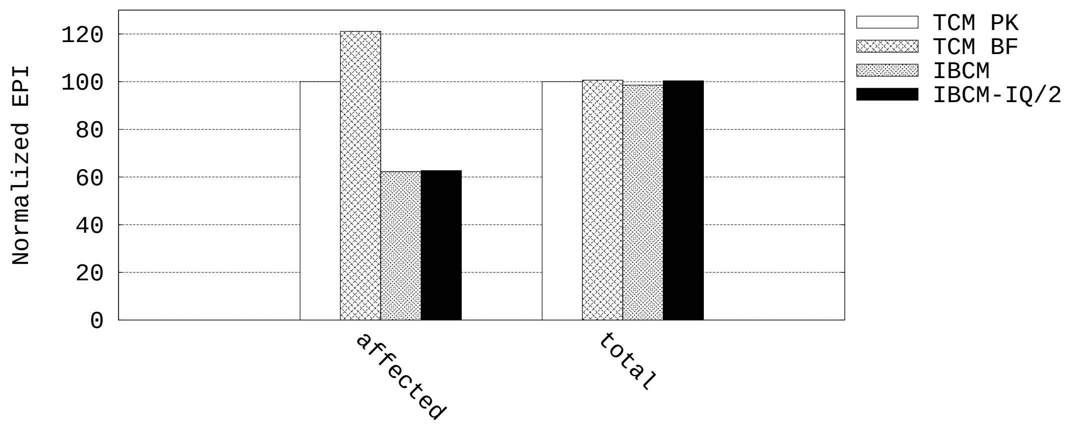

Fig. 4.7. Ideal configuration energy per instruction.

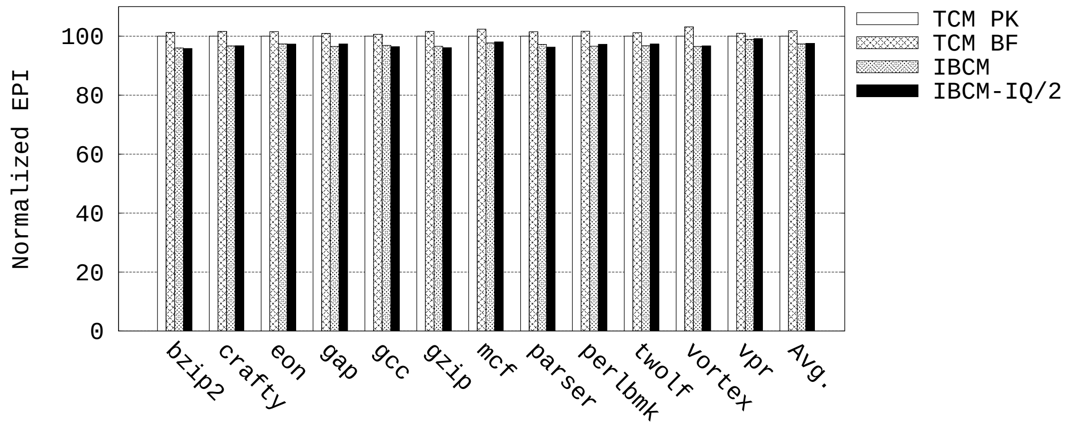

Fig. 4.8. Real configuration energy per instruction across SpecCPU benchmarks.

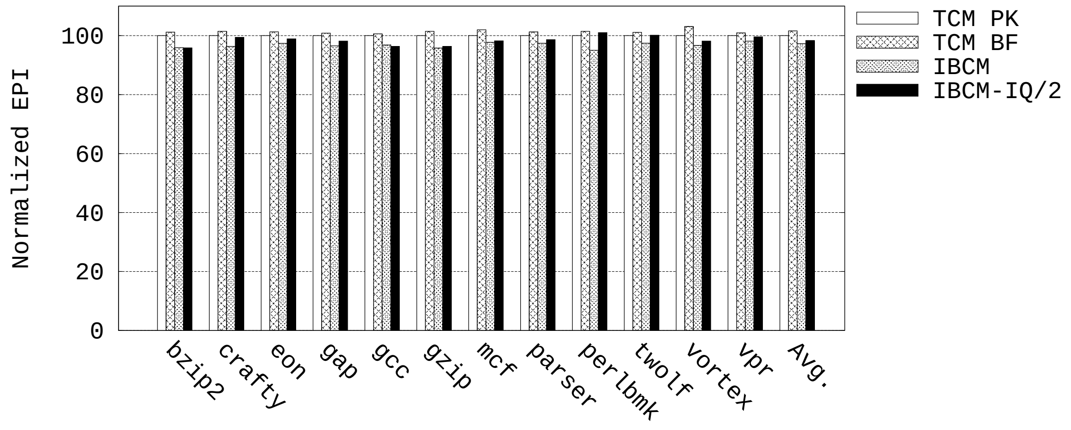

Fig. 4.9. Ideal configuration energy per instruction across SpecCPU benchmarks.

The expanded EPI is plotted in two graphs, Figure 4.8 for the realistic configuration and Figure 4.9 for the ideal processor configuration. These graphs (Figure 4.8 and Figure 4.9) are of EPI measurements over the “affected” region. Each graph contains thirteen sets of four bars. Each set corresponds to an integer benchmark in the SpecCPU 200 benchmark suite. The thirteenth bar, corresponds to the average across all twelve benchmarks. Bars inside each set correspond to different configurations: perfect-knowledge (TCM PK) configuration of *TCM*, brute-force (TCM BF) configuration of *TCM*, *IBCM* configuration, and *IBCM-IQ/2* configuration (issue queue is reduced by half).

Two observations can be drawn from Figure 4.6 and Figure 4.7. First, the ability to prevent unnecessary inter-cluster communication accounts for the EPI difference (in the affected region) between TCM PK and TCM BF. Yet, *IBCM* configurations are able to capture one additional advantage beyond eliminating unnecessary communication. *IBCM* reduces the EPI in the affected region by a significant amount. This is attributed to shorter intra-cluster result buses and saving duplicate register writes (albeit to a slightly smaller register file). Second, when considering the affected region in isolation the energy savings appear significant. Conversely, when considering the processor as a whole the energy savings appear insignificant. The trend is to consider energy savings over the whole processor; however, considering that the register files and ALUs (where the result-bus drivers are located) are among the “hottest” regions in a core, energy efficiency in these regions is a particularly respectable feature of *IBCM* [2].

In the expanded EPI graphs, Figure 4.8 and Figure 4.9, the effects of reduced communication are readily visible. *TCM* configurations expend energy bypassing to each functional unit in a cluster over the longer result buses; however, *IBCM*'s reduced communication (fully connected ALUs are on either side of the register file) expends less energy driving the shorter, less connected result buses. The variances in *IBCM* designs are attributed to load balancing, as described in Section 3.2.4, as well as the reduced issue queue size, which results in long dependency chains being split across groups. This will raise the communication, and energy consumption of the design. This variance in communication is readily visible in the ideal configuration (Figure 4.9) as there is less interference with *bypass fanout* as described in Section 2.1 because of the ideal resource configuration (specified in Table 4.1).

5. SUMMARY

First, this chapter discusses related works associated with *IBCM*. Second, future work on incomplete bypass-based designs is detailed. Finally, concluding remarks and summary of the results follow.

5.1 Related Work

Ahuja *et.al.* [10] characterize bypass network utilization in an in-order single-issue processor by measuring the activity on any given bypass path. This is not an appropriate metric for dynamically scheduled superscalar processors where a given static instruction may be issued to a different functional unit in each of its dynamic instances. The introduced metric of *bypass fanout* is more appropriate and maps directly to the length of result buses. Ahuja *et.al.* [10] also evaluated the performance trade-offs of incomplete bypassing for in-order, single-issue processors. This work focused on static techniques to transform the code in order to minimize the impact of incomplete bypassing.

In general, register caching [22] with early writeback to the register cache is a way to reduce the "levels" in the bypass network because the newly produced values are preserved in register caches even though they haven't been written back to the main register file. The bypass network simplification that results from register caching is orthogonal to *IBCM*since focus is on incomplete bypassing within a single stage (EX-EX).

Butts and Sohi [6] propose *degree-of-use* which serves many useful purposes, such as dead instruction removal; however, the metric of *bypass fanout* measures a subset of the total degree of use.

Park *et.al.* [23] have argued for reducing register port pressure by exploiting the fact that a large number of operands are sourced from the bypass network. In the common case, *IBCM* should not interfere with their technique as *IBCM* does not aim to replace bypass communication with register communication. Note, *bypass fanout* claims that each value is bypassed very few times; in addition, this claim does not contradict their claim that a large number of operands are sourced from the bypass network.

Aggarwal and Franklin examine instruction replication in hardware to minimize the performance loss due to clustering [16]. Aleta *et.al.* describes a compiler technique for instruction replication to minimize inter-cluster communication in clustered microarchitectures [17]. Similar approaches may be used in *IBCM* designs to replicate instructions across clusters.

5.2 Future Work

There are a few notable branches from this project. This work focuses on bypass wire delay and not other delay bottlenecks such as issue queues. This work may be incorporated into research that reduces the clock criticality of other pipeline stages, as well as the issue stage. Reducing other pipeline stages to sub-critical status, will make *IBCM* designs more attractive to processor performance.

The other direction lies in the scheduling of instructions on *IBCM* designs. As mentioned earlier, the body of work applicable to clustering is also applicable to *IBCM*, yet there is still room for tuning these policies to *IBCM* designs. The body of work on scheduling instructions shows the extent of variability programs and applications exert on processor architectures. Such heuristics as level of criticality, stall vs. steer, etc... allow new scheduler implementation with *IBCM* designs at the heart of the policy.

5.3 Conclusion

Single thread performance and energy efficiency remain important design goals for processor design. Significant progress has been made since the days of the monolithic superscalar processor design. This has been accomplished with the introduction of clustered architectures. This work goes beyond existing clustering implementations to propose a novel form of clustering — Incomplete Bypass-based Clustered Micro-architectures (IBCM) — that achieves 10% better performance than traditional clustered microarchitectures. The basic tradeoff of clustering is to have a slight sacrifice of ILP for disproportionate increase in clock-speed. The *IBCM* implementation goes farther along in this direction than *TCM*s. *IBCM* also has a slight sacrifice of ILP (2% reduction in IPC) compared to a *TCM*, but achieves 13% faster clock speed. Finally, *IBCM* offers improved energy efficiency in the register files and ALU output drivers. The improvement is modest when considering the processor as a whole, but their significance lies in the fact that the improvements in energy efficiency are in the hottest parts of the processor core. Overall, a novel design is proposed, which increases the performance while enhancing the benefits of clustered architectures. This performance also comes without much of the overhead associated with clustered architectures. Reducing needless communication that allows shorter wires, increases performance, while improving energy efficiency in the execution stage of a processor core are a boon to any design. *IBCM* also remains relevant as cores become simpler, so long as they remain capable of multiple issue. As general purpose computing moves to multicore/manycore designs, single core performance must not sustain neglect.

LIST OF REFERENCES

LIST OF REFERENCES

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[2] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Veusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 94–125, 2004.

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the complexity of superscalar processors," Tech. Rep. CSTR-96-1328, University of Wisconsin-Madison, November 1996.

[4] D. Burger and T. Austin, "The simplescalar tool set," tech. rep., University of Wisconsin-Madison, 1997.

[5] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer System*, June 2003.

[6] J. A. Butts and G. S. Sohi, "Characterizing and predicting value degree of use," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 15–26, IEEE Computer Society Press, 2002.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 83–94, June 2000.

[8] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE MICRO*, vol. 16, pp. 28–40, April 1996.

[9] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, Mar/Apr 1999.

[10] P. S. Ahuja, D. W. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 36–45, IEEE Computer Society Press, 1995.

[11] "S.i. association. international technology roadmap for semiconductors," 2006.

[12] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad-cluster, dynamically scheduled, superscalar processors," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, (New York, NY, USA), pp. 337–347, ACM Press, 2000.

[13] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, “Dynamic prediction of critical path instructions,” in *The Seventh International Symposium on High-Performance Computer Architecture*, pp. 185–195, 2001.

[14] B. Fields, S. Rubin, and R. Bodik, “Focusing processor policies via critical-path prediction,” in *ISCA '01: Proceedings of the 28th annual International Symposium on Computer Architecture*, pp. 74–85, 2001.

[15] P. Salverda and C. Zilles, “A criticality analysis of clustering in superscalar processors,” in *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 55–66, IEEE Computer Society, 2005.

[16] A. Aggarwal and M. Franklin, “Instruction replication: Reducing delays due to inter-pe communication latency,” in *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, p. 46, 2003.

[17] A. Alet, J. M. Codina, A. Gonzlez, and D. Kaeli, “Instruction replication for clustered microarchitectures,” in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 326, IEEE Computer Society, 2003.

[18] J. Stark, M. D. Brown, and Y. N. Patt, “On pipelining dynamic instruction scheduling logic,” in *International Symposium on Microarchitecture*, pp. 57–66, 2000.

[19] S. Standard Performance Evaluation Corporation, “The SPEC CPU2000 Benchmark Suite,” 2000.

[20] “A speculative control scheme for an energy-efficient banked register file,” *IEEE Trans. Comput.*, vol. 54, no. 6, pp. 741–751, 2005. Student Member-Jessica H. Tseng and Member-Krste Asanovic.

[21] P. G. Sassone and D. S. Wills, “Multicycle broadcast bypass: Too readily overlooked,” in *WCED ISCA '04: Proceedings of the Workshop on Complexity Effective Design*, 2004.

[22] J. Butts and G. Sohi, “Use-based register caching with decoupled indexing,” in *Proceedings of the 31st annual international symposium on Computer Architecture*, p. 302, IEEE Computer Society, 2004.

[23] I. Park, M. Powell, and T. N. Vijaykumar, “Reducing register ports for higher speed and lower power,” in *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, pp. 171–181, November 2002.