Abstract

GUPTA, NIKHIL

Slipstream-Based Steering for Clustered Microarchitectures

(Under the direction of Dr. Eric Rotenberg)

To harvest increasing levels of ILP while maintaining a fast clock, clustered microarchitectures have been proposed. However, the fast clock enabled by clustering comes at the cost of multiple cycles to communicate values among clusters. A chief performance limiter of a clustered microarchitecture is inter-cluster communication between instructions. Specifically, inter-cluster communication between critical-path instructions is the most harmful. The slipstream paradigm identifies critical-path instructions in the form of effectual instructions.

We propose eliminating virtually all inter-cluster communication among effectual instructions, simply by ensuring that the entire effectual component of the program executes within a cluster. This thesis proposes two execution models: the *replication model* and the *dedicated-cluster model*. In the replication model, a copy of the effectual component is executed on each of the clusters and the ineffectual instructions are shared among the clusters. In the dedicated-cluster model, the effectual component is executed on a single cluster (the *effectual cluster*), while all ineffectual instructions are steered to the remaining clusters. Outcomes of ineffectual instructions are not needed (in hindsight), hence their execution can be exposed to inter-cluster communication latency without significantly impacting overall performance.

IPC of the replication model on dual clusters and quad clusters is virtually independent of inter-cluster communication latency. IPC decreases by 1.3% and 0.8%,

on average, for a dual-cluster and quad-cluster microarchitecture, respectively, when inter-cluster communication latency increases from 2 cycles to 16 cycles. In contrast, IPC of the best-performing dependence-based steering decreases by 35% and 55%, on average, for a dual-cluster and quad-cluster microarchitecture, respectively, over the same latency range. For dual clusters and quad clusters with low latencies (fewer than 8 cycles), slipstream-based steering underperforms conventional steering because improved latency tolerance is outweighed by higher contention for execution bandwidth within clusters. However, the balance shifts at higher latencies. For a dual-cluster microarchitecture, dedicated-cluster-based steering outperforms the best conventional steering on average by 10% and 24% at 8 and 16 cycles, respectively. For a quad-cluster microarchitecture, replication-based steering outperforms the best conventional steering on average by 10% and 32% at 8 and 16 cycles, respectively.

Slipstream-based steering desensitizes the IPC performance of a clustered microarchitecture to tens of cycles of inter-cluster communication latency. As feature sizes shrink, it will take multiple cycles to propagate signals across the processor chip. For a clustered microarchitecture, this implies that with further scaling of feature size, the inter-cluster communication latency will increase to the point where microarchitects must manage a distributed system on a chip. Thus, if individual clusters are clocked faster, at the expense of increasing inter-cluster communication latency, performance of a clustered microarchitecture using slipstream-based steering will improve considerably as compared to a clustered microarchitecture using the best conventional steering approach.

Slipstream-Based Steering for Clustered Microarchitectures

by

Nikhil Gupta

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh

2003

Approved by

_____
Dr. Eric Rotenberg, Chair of Advisory Committee

_____
Dr. Gregory T. Byrd

_____
Dr. Thomas M. Conte

BIOGRAPHY

Nikhil Gupta was born on December 31, 1979, in Amritsar, India. He graduated with a Bachelor of Engineering (B.E.) degree in Instrumentation and Control Engineering from Nirma Institute of Technology (affiliated with Gujarat University), Ahmedabad, India, in June 2001.

In fall 2001, he joined the masters program in Computer Engineering at North Carolina State University, Raleigh, NC. He was part of the slipstream project and completed his masters thesis under the direction of Dr. Eric Rotenberg.

ACKNOWLEDGEMENT

I would like to dedicate this thesis to my parents Yash and Ashoki. My father, for his boundless (well almost) knowledge, which never ceases to amaze and inspire me. My mother, under whose meticulous care, life has been such a breeze. Their love and encouragement means a lot to me. My sister Richa, who with her effervescent smile lights up our family. I would like to thank Shubha for her love and constant support.

I would really like to thank Dr. Eric Rotenberg for having provided me with this opportunity to work with him. Working for him has been a pleasant and fantastic learning experience. He is easily one of the best teachers I have ever studied under (though he has not heard me say that). His ability to breakdown the most complex of problems and explain them even to a layman always amazes me. I have learnt a lot from him about computer architecture, better writing, and a good work ethic.

I would like to thank Dr. Greg Byrd and Dr. Tom Conte for agreeing to be on my Masters thesis committee.

I would also like to thank Sandy Bronson for all that she does to make the life of all of us students easy.

I would like to thank Aravindh Anantaraman, Karthik Sunadaramoorthy, Prakash Ramrakhyani, and Zach Purser for making the working environment (first) in EGRC 438

and (then) in Partners-I, as friendly as it could possibly be. Their invaluable insights and regular help that they provided on the simulator and other practical aspects was always very helpful. I would also like to thank Huiyang Zhou, Mark Toburen, Saurabh Sharma, and Ugur Gunal for providing a ready ear for my doubts.

I would also like to thank the technical support team of the Electrical and Computer Engineering department, who were always prompt in solving my problems.

I would like to thank Aaditya Goswami, Viraj Mehta, and Vikas Garg for having put up with my cooking, erratic behavior, bad jokes and an occasional mania for cleanliness, all of these two years. A special thanks to Chintan Trivedi, Harshit Shah, Manas Somaiya, Jinal Dalal, and Vishal Khanderia for making the transition of living in a new country, a new environment so easy. I have always valued their friendship and advice and will continue to do so in the future. A special thanks to Shalin Dalal, Magathi Jayaram, Rachana Shah, and Rachana Doshi for their friendship.

INDEX

List of Figures

List of Tables

1 Introduction

Microarchitects attempt to exploit higher levels of instruction-level parallelism (ILP) by developing processors with larger instruction windows and higher peak issue rates. Studies have shown that it is not possible to increase processor complexity without adversely affecting cycle time [2][18]. To harvest increasing levels of ILP while maintaining a fast clock, clustered microarchitectures have been proposed. A clustered microarchitecture breaks the monolithic execution window of a conventional superscalar processor into multiple smaller windows, called clusters. Each cluster contains a relatively small issue queue, a copy of the register file, a small number of dedicated function units, and short (i.e., fast) bypasses among its function units. Effectively, each cluster is a small-scale superscalar processor. The efficiency of individual clusters (less complex issue logic, fast local bypasses, fewer register file read ports, etc.) allows the clustered microarchitecture to be clocked faster than a monolithic microarchitecture. Because of its significant advantages, clustering has already been implemented in the Alpha 21264 [13].

However, the fast clock enabled by clustering comes at the cost of multiple cycles to communicate values among clusters. If producer and consumer instructions are routed to different clusters, the execution of consumer instructions will be delayed due to *inter-cluster communication*. Thus, performance in terms of instructions executed per cycle (IPC) is significantly affected by how instructions are steered, i.e., how instructions are assigned to clusters. Various steering schemes have been proposed. Many of these are

static (compiler-based) [9][17], while others are dynamic (based on run-time decisions) [4][6][7]. More recent steering approaches are based on critical-path prediction [10][30], the idea being that the execution of instructions on the critical path should not be delayed by global communication.

Instructions that constrain the execution time of a program constitute its critical path. To improve processor performance, the execution of critical-path instructions has to be optimized. In the case of a processor with a clustered microarchitecture, this implies that resource bottlenecks (e.g., limited function units in a cluster) and inter-cluster communication latency must have minimum impact on the execution of critical-path instructions.

This thesis proposes exploiting the distinction between effectual and ineffectual instructions, originally defined in the context of slipstream processors [14][20][21][24][28], in the context of clustered microarchitectures. The key idea put forth in the slipstream paradigm is that only some of the instructions in the dynamic instruction stream are needed for the correct forward progress of the program. These instructions are called *effectual* instructions. Instructions that are non-essential for correct forward progress are called *ineffectual* instructions. Ineffectual instructions include unreferenced writes, non-modifying writes, and highly predictable branches (and all of the computation chains feeding these instructions). We observe that ineffectual instructions are by definition not critical. Their outcomes are not needed, hence their completion does not need to be timely.

- *Unreferenced writes*: These instructions produce values that are never consumed.
- *Non-modifying writes*: These instructions do not change machine state and so effectual dependent instructions are not truly dependent on them.
- *Correctly-predicted branches*: Since the predictions are correct, verification can be deferred without penalty (other than tying up rename checkpoints).

We conclude that the effectual component of the program is critical and the ineffectual component is not, and this distinction should shape the way in which instructions are assigned to clusters on a clustered microarchitecture.

The chief performance limiter of a clustered microarchitecture is inter-cluster communication between instructions. Specifically, inter-cluster communication between critical-path instructions is the most harmful. The slipstream paradigm identifies critical-path instructions in the form of effectual instructions. We propose eliminating virtually all inter-cluster communication among effectual instructions, simply by ensuring that the entire effectual component of the program executes within a cluster. We propose two alternative execution models that achieve this.

- In the first model (called the *replication model*), the effectual program component (called the A-stream in slipstream processors [20]) is replicated and executed on all clusters. Ineffectual instructions are not replicated, i.e., they are executed in a distributed manner on all clusters. Consider the quad-cluster configuration shown in Figure 1-1. The dynamic instruction stream can be divided into an effectual

component and an ineffectual component. A full copy of the effectual component is executed on each cluster, whereas ineffectual instructions are distributed among the clusters.

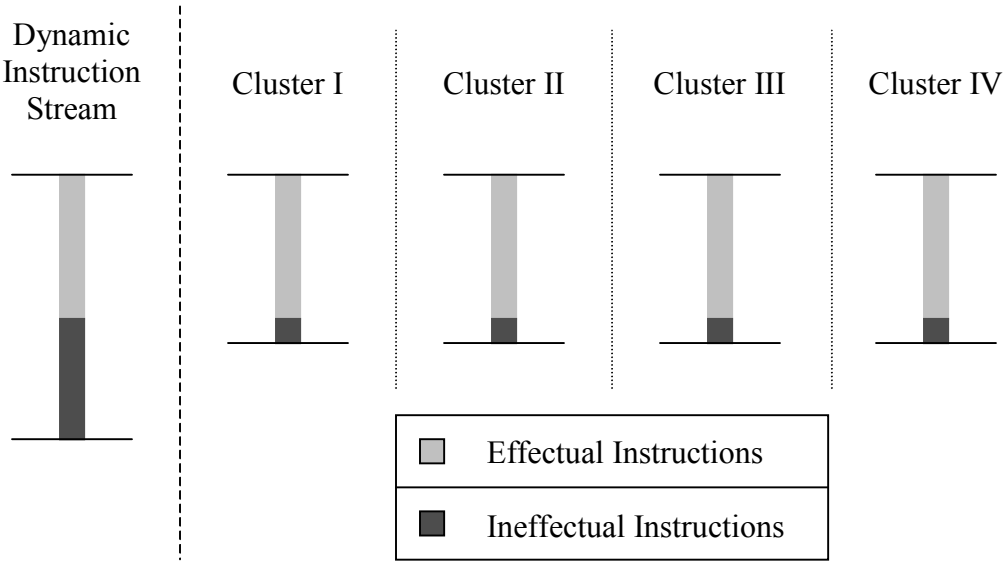

Figure 1-1: Replication model of instruction execution in a clustered microarchitecture.

- In the second model (called the *dedicated-cluster model*), a single copy of the effectual program component is executed on a dedicated cluster (i.e., no replication). Ineffectual instructions are executed in a distributed manner on all other clusters. From Figure 1-2, it can be seen that the effectual component of the dynamic instruction stream is executed only on cluster I, while ineffectual instructions are steered to clusters II, III, and IV.

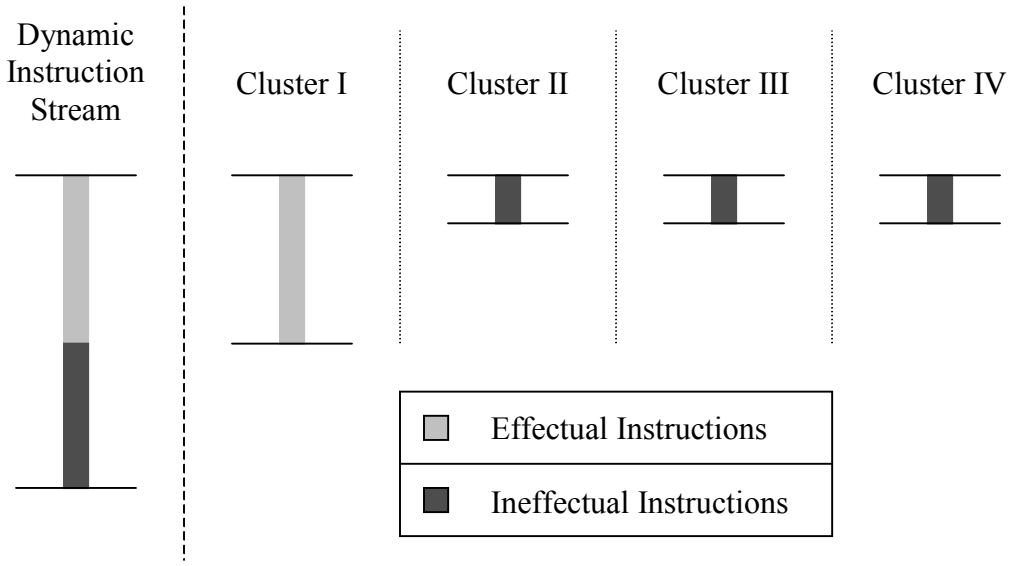

Figure 1-2: Dedicated-cluster model of instruction execution in a clustered microarchitecture.

1.1 Contributions

This thesis proposes methods for minimizing inter-cluster communication between effectual instructions, virtually desensitizing the performance of a clustered microarchitecture to inter-cluster communication latency. The contributions of this thesis are as follows.

- A clustered microarchitecture has been developed that incorporates slipstream components for distinguishing effectual and ineffectual instructions. The rename stage is enhanced with conventional steering heuristics and new heuristics, including assigning the same instruction to multiple clusters (replication) according to various execution models. A key innovation is leveraging the

existing renaming/register file mechanisms for supporting the new execution models transparently.

- Four new algorithms (two based on the replication model and two based on the dedicated-cluster model) for instruction execution in a clustered microarchitecture are developed.

  Replication of effectual component: A copy of the effectual program component is executed on each of the clusters. Thus, virtually no inter-cluster communication takes place between effectual instructions. The execution of ineffectual instructions is distributed across all clusters as the delay experienced in their execution can be tolerated.

  Replication of effectual component with store distribution: The first algorithm puts extra pressure on the limited issue bandwidth of a cluster. To ease this pressure, the algorithm is slightly modified. Effectual stores are distributed instead of replicated, like ineffectual instructions. The rationale is that effectual stores are potentially more latency-tolerant than register-writing instructions, due to typically longer separation between stores and loads as compared to producers and consumers of register values. Issue bandwidth is freed within each cluster because each cluster no longer executes all effectual stores (effectual stores, like ineffectual instructions, are shared equally among all clusters).

Dedicated-cluster for effectual component: The effectual program component is executed on a single dedicated cluster called the *effectual cluster*. Ineffectual instructions are steered to clusters other than the effectual cluster. No inter-cluster communication exists between effectual instructions as they are executed on a single cluster.

Dedicated-cluster for effectual component with store distribution: In the previous algorithm, a large number of instructions are executed on the effectual cluster as compared to the number of instructions executed on each of the other clusters. Off-loading effectual stores to other clusters reduces demand on the effectual cluster and does not increase inter-cluster register communication. All effectual instructions except effectual stores are executed on a single cluster, while effectual stores and all ineffectual instructions are steered to the remaining clusters. The speedup gained by freeing execution resources on the effectual cluster compensates for the increase in inter-cluster communication between effectual stores and loads. This is especially beneficial in the case of low inter-cluster bypass latencies, where execution bandwidth, not communication, is the bottleneck.

- We study the effect of increasing inter-cluster communication latency on the performance of various steering heuristics for dual-cluster and quad-cluster configurations. To the best of our knowledge, no other research considers the impact of more than two cycles of inter-cluster communication latency.

(However, Aggarwal and Franklin [1] indirectly observed the effect of increasing inter-cluster communication latency by increasing the number of clusters.) We are interested in the impact of ramping up the clock rate of each cluster to the point where inter-cluster communication is in the tens-of-cycles regime. This scenario is effectively a distributed system on a single chip.

- We evaluate slipstream-related parameters in the design space of slipstream-based clustered microarchitectures. We vary instruction-removal confidence threshold and instruction-removal criteria.

1.2 Organization of the Thesis

Chapter 2 gives an overview of clustered microarchitectures and conventional steering heuristics. Chapter 3 gives an overview of slipstream components that separate effectual and ineffectual instructions. In Chapter 4, slipstream-based steering models are proposed and explained. The simulation methodology and benchmarks are described in Chapter 5. Chapter 6 presents the experimental results. Related work is discussed in Chapter 7. Chapter 8 summarizes the thesis and proposes future work.

2 Processors with a Clustered Microarchitecture

The amount of instruction-level parallelism (ILP) that can be extracted from a program plays a major role in the performance of a superscalar processor. Larger instruction windows combined with higher peak issue rates and more function units can exploit large amounts of ILP. Studies have shown that large instruction windows and high issue rates result in slower wakeup and select logic [2][18]. Also, larger register files and more function units result in slower bypasses. Therefore, it is not possible to build large monolithic instruction windows without adversely affecting the cycle time of a processor.

In order to harvest increasing levels of ILP while maintaining a fast clock, clustered microarchitectures have been proposed. A clustered microarchitecture divides the monolithic execution window of a conventional superscalar processor into multiple smaller windows called clusters. A cluster consists of a relatively small instruction window, a copy of the register file, a small number of dedicated function units, and short bypasses among its function units called *intra-cluster bypasses*. Intra-cluster bypasses are fast because they span only a few function units. A global bypass bus called the *inter-cluster bypass bus* interconnects all the clusters in the processor. The longer inter-cluster bypasses are slow because they span all clusters.

The method by which instructions are assigned to a specific cluster for execution is called *steering*. The pipeline stage that performs steering depends on the type of

clustered microarchitecture. For the type of clustered microarchitecture described earlier, the dispatch stage performs steering. This has been called *dispatch-driven* instruction steering by Parlacharla and Smith [18]. Dispatch-driven steering results in a highly-efficient design for individual clusters. The smaller instruction window and moderate issue bandwidth imply less complex (i.e., fast) wakeup and select logic. Fewer function units and a smaller register file (due to fewer read ports) enable the use of shorter (i.e., faster) bypasses within a cluster. Thus, a clustered microarchitecture can be clocked faster than a monolithic microarchitecture.

A second form of clustered microarchitecture uses a monolithic instruction window that issues instructions to multiple clusters. In this case, a cluster only has a copy of the register file, a small number of dedicated function units, and short bypasses among its function units. As before, inter-cluster bypasses connect the clusters and are slower than the fast intra-cluster bypasses. In this form of clustered microarchitecture, an instruction resides in the monolithic issue queue and is assigned to a cluster only when it becomes ready to issue. Thus, instruction steering is performed in the issue stage. This approach has been called *execution-driven* instruction steering by Parlacharla and Smith [18]. This form of clustered microarchitecture has been implemented in the integer pipeline of the Alpha 21264 [13]. The integer pipeline has a single issue queue with two clusters. Each integer cluster consists of a copy of the integer register file and an equal number of integer function units. A value produced in a cluster is communicated to consumers in the same cluster, in the same cycle the value is produced. However, the value takes an extra cycle to be communicated to consumers in the other cluster.

Execution-driven steering tends to be more effective than dispatch-driven steering because it defers steering until a more informed decision can be made. However, this approach uses a large instruction window and is therefore less complexity-effective. Also, steering logic is added to the issue stage, making it even more complex. Therefore, the dispatch-driven approach is the better method in terms of efficiency.

2.1 Clustered Microarchitecture

In this thesis, we will consider a clustered microarchitecture that utilizes dispatch-driven steering. Instructions are steered to clusters at the time of dispatch. Consequently, instructions issue and execute within the cluster to which they are steered, as they would on a small-scale superscalar processor. A value produced within a cluster is quickly bypassed to consumers in the same cluster, via the intra-cluster bypasses. Thus, producers and consumers execute in consecutive cycles if they are in the same cluster. All values produced within a cluster are also communicated to the other clusters. However, values produced in a cluster are available to consumers in another cluster only after a certain delay called the *inter-cluster communication latency*. This delay depends on the number of clusters and the size of the clusters.

As we move from a centralized microarchitecture to a clustered microarchitecture, several changes are needed. The salient features of a clustered microarchitecture are highlighted by comparing a quad-clustered configuration to a non-clustered configuration, in Figure 2-1.

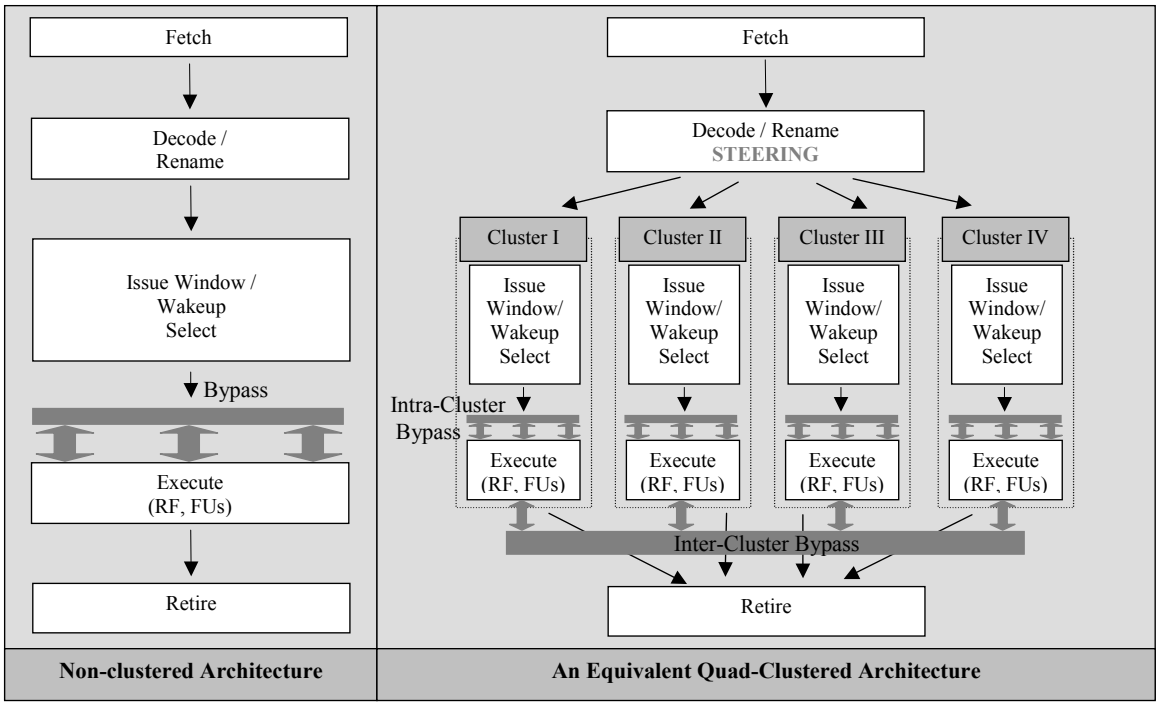

Figure 2-1: Comparison between a non-clustered microarchitecture and a quad-clustered microarchitecture.

Note that the execution resources of the monolithic microarchitecture are divided equally among all clusters. From the figure, the following observations can be made.

- Each cluster has a copy of the physical register file. A value produced within a cluster is written into its copy of the register file, and all other copies. However, it takes longer for the value to propagate to other copies of the register file (i.e., register files in other clusters), and it is dictated by the inter-cluster bypass latency. The crucial advantage of replicating the register file is that the number of read ports for each copy of the register file is far fewer than the number of read ports for the register file of a monolithic configuration. The reason is that fewer instructions issue each cycle within a cluster.
- The issue window, issue bandwidth, and function units of the monolithic configuration are divided equally among all clusters.

- An intra-cluster bypass bus is used to communicate values within a cluster and an inter-cluster bypass bus is added to connect all clusters. The intra-cluster bypasses are much faster than the inter-cluster bypass.
- To ensure that store-load dependences are observed, the memory disambiguation unit is replicated, i.e., all clusters maintain a copy of the load/store queue. Stores broadcast their addresses and values to all clusters, incurring the penalty of inter-cluster communication latency. The store addresses and values are used by each cluster to detect memory dependences and perform store-load forwarding as needed.
- There is no change to the in-order front-end (fetch unit, decode, and rename unit) and in-order back-end (retirement unit). The fetch, dispatch, and retire stages remain centralized. Only the out-of-order execution window is distributed. The only modification is that steering functionality is added to the dispatch stage.

Table 2-1 summarizes how the various resources of a non-clustered microarchitecture change after distributing them among clusters in a quad-clustered configuration. The first column in the table shows the hardware resources. The second column characterizes the resources for the monolithic configuration. The third column characterizes the resources for the quad-cluster configuration. From the table, the following observations can be made regarding a clustered microarchitecture.

- The fetch and dispatch stages are common to all clusters, hence the fetch bandwidth and dispatch bandwidth are not distributed. Likewise, the retirement stage is common to all clusters.

- The organization of the execution window is effectively transparent to renaming and retirement, hence there is only one architectural map table, rename map table, and free list, as usual.
- The issue bandwidth, cache ports, instruction queue, and function units of the monolithic microarchitecture are divided equally among the four clusters.
- The register file is replicated, one copy per cluster. However, the number of read ports to each cluster register file is reduced with respect to the monolithic register file, since each cluster register file only needs to support reads from instructions within the cluster. The number of write ports to each cluster register file is not reduced with respect to the monolithic register file because all values are written into each cluster register file.
- The load/store queue is replicated, one copy per cluster, as described earlier.

Table 2-1: Comparison of hardware resources between a base superscalar processor and a quad-clustered processor.

Hardware Resource	Base Superscalar	Quad – Cluster
Fetch bandwidth	8	8 (common to all)
Dispatch bandwidth	8	8 (common to all)
Active list size	128	128 (common to all)
Free list size	128	128 (common to all)
Rename map table	1	1 (common to all)
Architectural map table	1	1 (common to all)
Issue bandwidth	*8*	*2 per cluster*
Cache ports	*4*	*1 per cluster*
Issue queue	*128*	*32 per cluster*
Registers	*195*	*195 per cluster*
Register read ports	*16*	*4 per cluster*
Register write ports	*8*	*8 per cluster*
Load / store unit	*1*	*1 per cluster*
Retire bandwidth	8	8 (common to all)

Each cluster has a copy of the physical register file. This not only provides fast local access, but it also reduces the number of read ports to the register file. The number of read ports affects the size and hence the speed of the register file. This gives the clustered microarchitecture an edge over the non-clustered microarchitecture. The number of read and write ports are calculated in the following two sections.

2.1.1 Number of write ports to a cluster register file

An instruction can produce only one value. Thus, a single write port is needed per instruction per cycle. Therefore, the maximum number of write ports (WP) needed for a monolithic microarchitecture is as follows.

$$WP_{monolithic} = IW \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad \textbf{Equation 2-1}$$

The number of writes that can occur simultaneously within a cluster is equal to the aggregate issue width of the clustered microarchitecture. The reason is that all values have to be written in each cluster register file, even those produced by other clusters. Thus, the number of write ports is determined by the aggregate issue width of the clustered microarchitecture.

$$WP_{cluster} = n * CIW \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad \textbf{Equation 2-2}$$

Above, n is the number of clusters and CIW is the cluster issue width.

The aggregate issue width of the clustered microarchitecture equals the total issue width of the monolithic microarchitecture, therefore:

$$n * CIW = IW \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad \textbf{Equation 2-3}$$

$WP_{cluster} = IW$ …… **Equation 2-4**

$WP_{cluster} = WP_{monolithic}$ …… **Equation 2-5**

Thus, the number of write ports to the cluster register file is equal to the number of write ports to the monolithic register file.

2.1.2 Number of read ports to a cluster register file

In general, the number of read ports to a register file is determined by the issue width (IW) of the processor. An instruction can have a maximum of two source operands. Hence, two reads are performed at the same time for one instruction, which implies that two read ports are needed per instruction per cycle. Therefore, the maximum number of read ports (RP) needed for a monolithic microarchitecture is as follows.

$RP_{monolithic} = 2 * IW$ …… **Equation 2-6**

The number of instructions that can be issued per cycle within a cluster is the cluster issue width (CIW). The number of read ports to the cluster register file is determined by this cluster issue width, as follows.

$RP_{cluster} = 2 * CIW$ …… **Equation 2-7**

$RP_{cluster} = 2 * IW / n$ …… **Equation 2-8**

$RP_{cluster} = RP_{monolithic} / n$ …… **Equation 2-9**

It can be seen that the number of read ports per cluster register file is reduced by a factor of n with respect to the monolithic register file, thereby reducing its complexity and improving the register access time. For the quad-cluster configuration shown,

$RP_{cluster} = ¼\ RP_{monolithic}$ …… **Equation 2-10**

2.2 Bottlenecks in a Clustered Microarchitecture

The efficiency of various hardware resources enables a clustered microarchitecture to have a shorter clock period than a monolithic microarchitecture. However, clustered microarchitectures suffer from the following bottlenecks.

- *Inter-Cluster Communication Latency*: It takes multiple clock cycles to communicate values via the long inter-cluster bypasses. The execution of an instruction whose source operands are produced in another cluster is delayed due to the extra time it takes for values to be communicated via the inter-cluster bypass.
- *Smaller Cluster Issue Bandwidth*: There may be more instructions ready for issue on a cluster than can be issued in a cycle, while at the same time, there may be empty issue slots available in another cluster. This imbalance leads to issue bandwidth stalls on one cluster and under-utilization of resources on the other.

2.3 Steering Heuristics

Steering heuristics dictate the performance achieved by a clustered microarchitecture. Effective steering can reduce inter-cluster communication and optimize resource usage, whereas ineffective steering can cause significant performance degradation and negate the advantage that a clustered microarchitecture provides over a monolithic microarchitecture. Various steering heuristics have been proposed.

Modulo$_n$ (Mod3) [4]: A new cluster is selected every n instructions on the basis of a round-robin policy. For example, for **Mod3** steering in a quad-cluster configuration, the first three instructions are assigned to cluster I, the next three to cluster II, and so on. This algorithm does not try to minimize inter-cluster communication but achieves a fairly good load balance via equal instruction distribution among clusters.

Branch-Cut (BC) [4]: Instructions are assigned to the same cluster until a branch is reached. Instructions after the branch are then assigned to a new cluster according to a round-robin policy. Therefore, instructions are steered on a basic block level. It has been observed that instructions within a basic block typically belong to the same dependence chain(s). Thus, forcing a possible dependent chain of instructions onto the same cluster would reduce inter-cluster communication and achieve a balanced distribution of instructions among the clusters.

Load-Cut (LC) [4]: Instructions are assigned to the same cluster until a load is reached. The load and instructions following the load are assigned to a new cluster according to a round-robin policy. Loads often begin a chain of dependent instructions. Thus, by changing the cluster on encountering a load, a possible dependent chain of instructions is steered to the same cluster and inter-cluster dependences are reduced accordingly.

Least-Loaded (LL) [4]**:** This algorithm tries to balance the distribution of instructions among clusters. Every new instruction is assigned to the *least-loaded cluster*. The least loaded cluster is the cluster with the least number of instructions in its issue queue.

Dependence-Based (Dep0): This steering heuristic was proposed by Canal et. al. [6]. An explicit attempt is made to minimize inter-cluster communication by steering consumer instructions to the same cluster as their producers. The information needed for steering is obtained during instruction decoding/renaming because register dependences have to be taken into consideration. The algorithm works as follows.

- If an instruction has no source operands, it is assigned to the least-loaded cluster.
- If an instruction has one source operand, it is assigned to the cluster where the source operand has been produced or will be produced.
- If an instruction has two source operands, and both are in the same cluster, then the instruction is assigned to that cluster. If the two source operands are produced in different clusters, then the instruction is assigned to the least-loaded cluster among the producer clusters.

Readiness-Dependence-Based (DepR): This steering heuristic is very similar to the one proposed by Baniasadi et. al. [4]. Like the previous dependence-based scheme, this scheme also explicitly tries to minimize inter-cluster communication, but it takes the readiness of the source operands into account. A source operand is said to be *globally ready* if it is ready for consumption in all the clusters, and not just its producer cluster. The algorithm works as follows.

- If an instruction has no source operands, it is assigned to the least-loaded cluster.
- If an instruction has one source operand and that operand is globally ready, it is assigned to the least-loaded cluster. Otherwise, the instruction is steered to the cluster that produced or will produce the source operand.
- If an instruction has two source operands and both are globally ready, it is assigned to the least-loaded cluster. Otherwise, it is assigned to the cluster that produces the youngest source. The youngest source refers to the source operand whose producer is the closest to this instruction (consumer), in program order.

3 Slipstream Components used for Steering

The slipstream paradigm [20] proposes that only a fraction of the dynamic instruction stream is needed for a program to make correct forward progress. This component of the dynamic instruction stream is termed *effectual*. Many general-purpose programs contain a significant number of instruction sequences that either have no effect on the final outcome of a program or are highly predictable. Such instructions are called *ineffectual* instructions. Ineffectual instructions include unreferenced writes, non-modifying writes, highly-predictable branches, and computation chains leading up to them.

A slipstream processor runs two redundant copies of a program, one slightly ahead of the other, on a chip multiprocessor (CMP) or a simultaneous multithreading (SMT) processor. Ineffectual instructions are speculatively removed from the leading program, called the advanced stream (A-stream). The A-stream is sped up because it fetches and executes fewer instructions than the original program. All data and control outcomes from the A-stream are communicated to the trailing program, called the redundant stream (R-stream). The R-stream compares the communicated outcomes against its own outcomes. If a deviation is detected, the corrupted A-stream context is recovered from the R-stream context. This deviation is called an *IR-misprediction*. The R-stream also exploits the outcomes of the A-stream as accurate branch and value predictions. Thus, although the R-stream retires the same number of instructions as the

original program, it fetches and executes much more efficiently. As a result, both program copies finish sooner than the original program.

Two slipstream components are needed for identifying ineffectual instructions.

1. The *instruction-removal predictor*, or *IR-predictor*, is essentially a branch predictor augmented for instruction removal. It generates the program counter (PC) for the next block of instructions to be fetched in the A-stream, similar to a conventional branch predictor. The IR-predictor also specifies a bit-vector that identifies ineffectual instructions within a fetch block.
2. The *instruction-removal detector*, or *IR-detector*, identifies past instructions which were not essential for the R-stream's correct forward progress. The IR-detector then conveys to the IR-predictor that these instructions can potentially be skipped in the A-stream, in the future. The IR-predictor removes the corresponding instructions from the A-stream after repeated indications by the IR-detector, i.e., after a certain confidence threshold has been reached.

3.1 IR-predictor

The IR-predictor is a conventional branch predictor augmented to keep track of instruction removal information. It is indexed like the *gshare* predictor[15], by XORing the PC with the global branch history bits. Each table entry contains the following information for a dynamic basic block.

- *Tag*: This is the start PC of the basic block and is used to determine whether the entry contains information for the block being fetched. A partial tag can be used to reduce the total storage, if predictor aliasing is negligible.
- *2-bit counter*: A 2-bit counter is used to predict the outcome of a basic block that ends with a conditional branch.
- *Confidence counters*: A resetting confidence counter [12] is provided for each instruction in the basic block. The counter corresponding to a particular instruction is incremented if the IR-detector identified that instruction to be ineffectual. Otherwise, the counter is reset to zero. Repeated indications by the IR-detector saturate the confidence counter, in which case the corresponding instruction is predicted to be ineffectual by the IR-predictor and removed from the A-stream when it is next encountered.

3.2 IR-detector

The IR-detector consumes retired R-stream instructions and data. It then identifies instructions which were not essential for correct forward progress, in retrospect. The IR-detector watches for any of the following three triggering conditions for instruction removal.

- *Unreferenced writes*, i.e., a write followed by a write to the same location, with no intervening read.
- *Non-modifying writes*, i.e., a write that does not modify the value of a location.
- *Correctly-predicted branches*.

When any of the above conditions are observed, the corresponding instruction is selected for removal and this information is passed on to the IR-predictor. Additional ineffectual instructions are selected by a technique called *back-propagation*. Back-propagation detects computation chains that feed the instructions selected for removal based on the triggering conditions mentioned above. An instruction can be selected for removal if all of its dependent instructions are selected for removal. For example, once a branch is selected, the computation leading to that branch is no longer needed and can be selected for removal, if no other instructions depend on the computation.

3.3 Use of IR-detector/IR-predictor in Steering Mechanisms

For our purposes, the IR-detector and IR-predictor are used only as a means for identifying predicted-ineffectual instructions. That is, they are not actually used to remove instructions, since there is no separate A-stream thread.

The bit vector produced by the IR-predictor identifies ineffectual instructions and this is used by the steering mechanism directly, for distinguishing between effectual and ineffectual instructions.

4 Thesis Contribution: Slipstream-Based Steering

The goal of slipstream-based steering is to desensitize IPC performance to tens of cycles of inter-cluster communication latency. We conjecture that executing the effectual component entirely on a single cluster will achieve this goal while still effectively exploiting the parallel resources available in the clustered microarchitecture.

4.1 Replication of Effectual Component (Rep0)

We attempt to eliminate inter-cluster communication among effectual instructions by executing a copy of the effectual component on each of the clusters. On the other hand, the ineffectual component of the program is distributed (shared) among the clusters. Thus, the only potential inter-cluster communication is between ineffectual instructions in different clusters. Intra-cluster communication is guaranteed between (1) effectual producers and effectual consumers, and (2) effectual producers and ineffectual consumers. The latter aspect distinguishes the replication models described in this section and Section 4.2 from the non-replication models (dedicated-cluster models) described in Sections 4.3 and 4.4.

To manage the distribution of ineffectual instructions, the dynamic instruction stream is divided into traces. A trace consists of one or more basic blocks and is composed of both effectual and ineffectual instructions. This can be seen in Figure 4-1 (light gray = effectual, dark gray = ineffectual). A, B, C, and D are traces that are part of

the dynamic instruction stream. C_{eff} is the predicted-effectual component and C_{ineff} is the predicted-ineffectual component of trace C. We use C_{br} to refer to all the branches in trace C. Branches can also be divided into predicted-effectual branches ($C_{br,eff}$) and predicted-ineffectual branches ($C_{br,ineff}$).

$$C = C_{eff} + C_{ineff} \quad \text{.....} \quad \textbf{Equation 4-1}$$

$$C_{br} = C_{br,eff} + C_{br,ineff} \quad \text{.....} \quad \textbf{Equation 4-2}$$

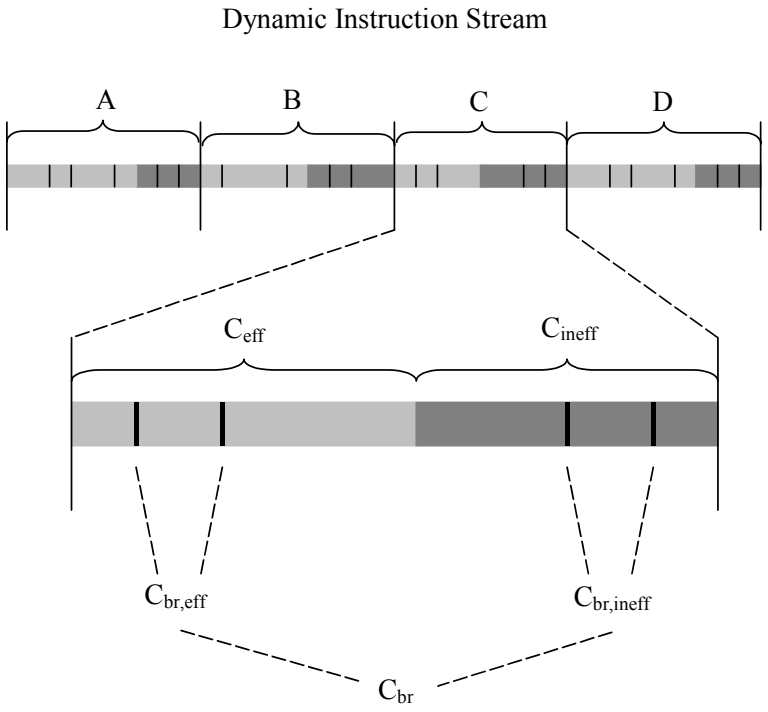

Figure 4-1: A breakdown of the dynamic instruction stream.

We say that each trace is *owned* by a particular cluster. A new trace is assigned to a cluster in a round-robin manner. For an n-cluster configuration, each cluster owns one out of every n traces. All instructions in a trace are executed within the cluster that owns the trace. Therefore, predicted-effectual and predicted-ineffectual instructions of a trace

are executed within the cluster which owns that trace. Only the predicted-effectual instructions of the trace are executed on clusters that do not own the trace. In this way, predicted-effectual instructions are replicated and predicted-ineffectual instructions are distributed equally among the clusters.

According to this policy, effectual branches are redundantly executed on all clusters, raising the question of how to resolve mispredicted branches. The first option is to exploit redundancy and thereby allow the earliest resolved branch to redirect the fetch unit. However, we found the complexity of managing multiple branches with a single rename checkpoint to be cumbersome. So, we instead implemented the second option, which is to steer effectual branches to only a single cluster, the one that owns the trace containing the branch. This also reduces contention for issue bandwidth within the clusters slightly. Note also that it does not increase inter-cluster communication.

Figure 4-2 shows how this heuristic works for a quad-cluster configuration. A segment of the dynamic instruction stream is divided into four traces, A, B, C, and D. A' is the reduced version of trace A. For the quad-cluster configuration shown, cluster I owns trace A, cluster II owns trace B, cluster III owns trace C, and cluster IV owns trace D. Thus, a cluster owns one in every four traces. All instructions in trace A are executed on cluster I, but predicted-ineffectual instructions and branches of trace A are not executed on clusters II, III, and IV. In other words, predicted-effectual instructions minus predicted-effectual branches of trace A are executed on clusters II, III, and IV. The reduced version of trace A can therefore be represented as follows.

$$A' = A - A_{ineff} - A_{br,eff} \quad \text{.....} \quad \textbf{Equation 4-3}$$

Since $A = A_{eff} + A_{ineff,}$

$$A' = A_{eff} - A_{br,eff} \quad \text{.....} \quad \textbf{Equation 4-4}$$

The same logic can be applied to the other traces.

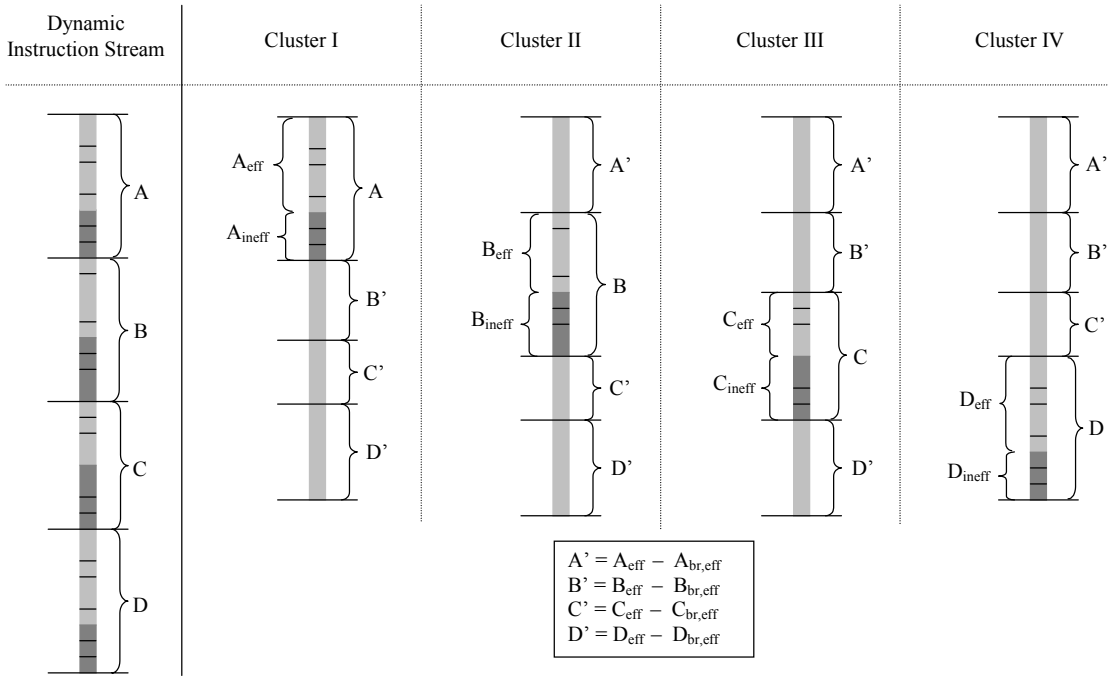

Figure 4-2: Replication of the effectual component.

4.1.1 Implications of redundant execution

In a clustered microarchitecture, each cluster has a copy of the physical register file. While decoding and renaming an instruction, the logical destination register is mapped to the same physical register on all clusters. Using our steering algorithm, a predicted-effectual instruction is dispatched to each of the clusters for redundant execution. This does not require changes to the existing rename mechanism. All copies of

an instruction will write their *identical* values to the same location in all the physical register files. Note that it is wasteful for effectual copies to broadcast their values to other clusters. We could eliminate communication for these redundant values, and thereby reduce register file write ports, reduce power consumption on inter-cluster bypasses, etc. This aspect is discussed in the future work section (Section 8.2).

As predicted-effectual instructions are executed redundantly on each of the clusters, it is possible that the result of a predicted-effectual instruction executed on cluster I is communicated to cluster II before cluster II's copy of the instruction has executed. In this case, dependent instructions in cluster II may issue/execute before the parent instruction executes in cluster II. Since the values produced by redundant instructions are identical, there is no problem with this scenario.

4.1.2 Regarding memory disambiguation

In the base clustered microarchitecture, broadcast of store addresses and values to other clusters is delayed by the inter-cluster communication latency. This may make memory disambiguation a major bottleneck due to loads conservatively waiting for all prior store addresses. A solution is to use aggressive memory dependence speculation. It has been shown that memory dependence speculation is almost as accurate as oracle memory disambiguation [8][16]. In any case, our architecture is less sensitive to this problem since effectual stores and loads reside in the same cluster. There is still the problem of ineffectual stores delaying disambiguation of effectual loads in other clusters. Memory dependence speculation could very accurately predict that there are no

dependences in this situation, since presumably effectual loads should not depend on ineffectual stores.

4.1.3 Implication of IR-mispredictions

An instruction-removal misprediction, or IR-misprediction, occurs when an effectual instruction is incorrectly predicted as an ineffectual instruction [20]. In our context, this implies that an instruction that should have executed redundantly on each of the clusters is now executed on only one of the clusters. Thus, effectual instructions that depend on an IR-mispredicted-instruction in another cluster will have to wait for the value to be communicated via the inter-cluster bypass. Therefore, an IR-misprediction results in inter-cluster communication between effectual instructions.

4.1.4 Changes to the microarchitecture

Key changes that must be introduced in the microarchitecture for implementing this scheme are as follows.

- After fetching an instruction, it is dispatched to all clusters if it is predicted-effectual (except for predicted-effectual branches), or steered to one of the clusters (depending on the owner of the trace) if it is predicted-ineffectual or a predicted-effectual branch.
- Instruction dispatch is stalled if any of the cluster issue queues are full. This is only true for predicted-effectual instructions (except branches) since they must be replicated.

- A predicted-effectual instruction is retired only after all copies have completed execution on each of the clusters. In other words, multiple instances of an instruction are re-integrated into a single instance before retiring.

4.2 Replication of Effectual Component with Store Distribution (RepS)

We modify the algorithm in Section 4.1 to reduce contention for cluster issue bandwidth. Instead of replicating effectual stores, we distribute them just like ineffectual instructions, based on the fact that memory dependence distances are longer than register dependence distances. Therefore, all predicted-effectual instructions except predicted-effectual branches and predicted-effectual stores are replicated. This implies that no branches or stores are replicated.

Distributing effectual stores increases inter-cluster communication between effectual stores and loads. On the other hand, by not executing stores redundantly, resources are freed for executing the residual effectual component faster. The newly-exposed inter-cluster communication between effectual stores and loads may be offset by the speedup gained by freeing execution resources in each cluster.

Figure 4-3 shows a segment of the dynamic instruction stream with a breakdown of effectual and ineffectual instructions. In addition to the terms used in Figure 4-1, a few

more terms are introduced. C_{st} indicates all the stores in trace C, $C_{st,eff}$ indicates the predicted-effectual stores, and $C_{st,ineff}$ indicates the predicted-ineffectual stores.

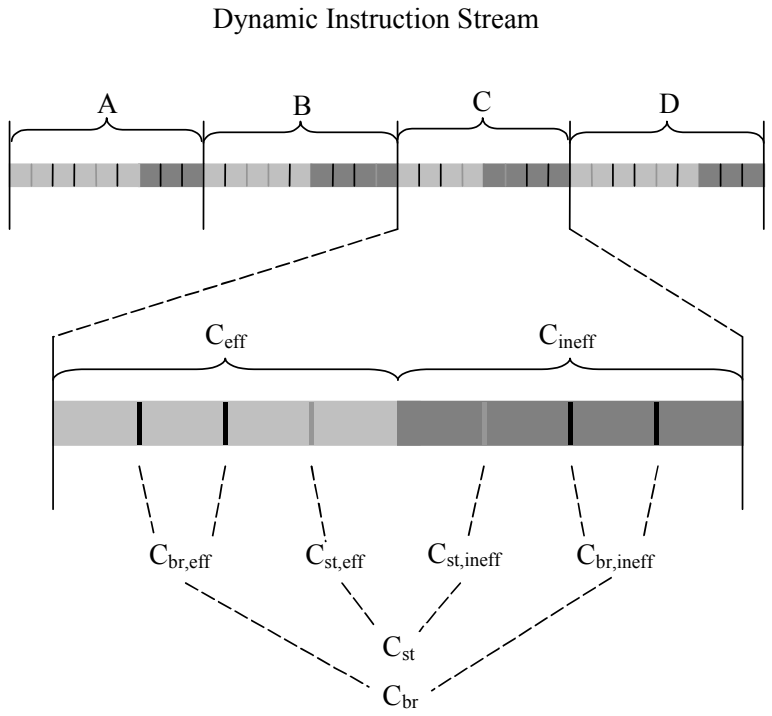

Figure 4-3: A breakdown of the dynamic instruction stream.

Consider the quad-cluster configuration shown in Figure 4-4. A segment of the dynamic instruction stream is divided into traces A, B, C, and D. A'' indicates the reduced version of a trace. As in the first algorithm, a cluster owns one in every four traces. Therefore, cluster I owns trace A, cluster II owns trace B, and so on. A cluster that owns a trace executes all instructions in that trace. On the other hand, a cluster that does not own a trace does not execute the branches, stores, and predicted-ineffectual instructions of that trace. In other words, it executes all predicted-effectual instructions

other than predicted-effectual stores and predicted-effectual branches, shown below for A".

$$A'' = A - A_{ineff} - A_{br,eff} - A_{st,eff} \quad \ldots\ldots\ldots\ldots\ldots \quad \textbf{Equation 4-5}$$

$$A'' = A_{eff} - A_{br,eff} - A_{st,eff} \quad \ldots\ldots\ldots\ldots\ldots \quad \textbf{Equation 4-6}$$

The same logic applies to other traces.

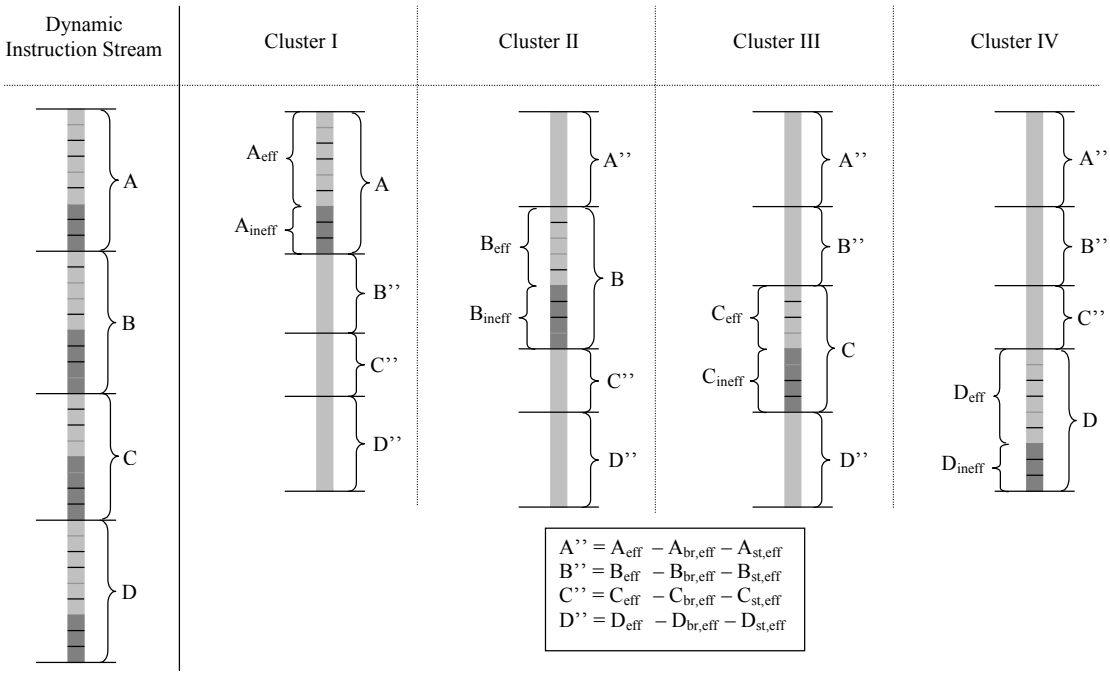

Figure 4-4: Replication of effectual component with store distribution.

Below are the key changes that must be introduced in the microarchitecture for implementing this scheme (in addition to the changes described for the previous scheme).

- After fetching an instruction, it is dispatched to all clusters if it is predicted-effectual (except for predicted-effectual branches/stores), or steered to one of the clusters (depending on the owner of the trace) if it is predicted-ineffectual or a predicted-effectual branch/store.

4.3 Dedicated Cluster for Effectual Component (DEC0)

We propose another model for eliminating inter-cluster communication between effectual instructions: All predicted-effectual instructions are executed on a single dedicated cluster and all predicted-ineffectual instructions are steered to clusters other than the dedicated cluster to which effectual instructions are steered. In this way, inter-cluster communication between effectual instructions is eliminated, except in the rare case of an IR-misprediction (since an effectual instruction is misclassified as an ineffectual instruction and steered to a different cluster). Ineffectual instructions, which depend on both effectual and ineffectual instructions, experience delays in execution due to inter-cluster communication latency. As these instructions are less critical, these delays can be tolerated. Note that the advantage of this model with respect to the previous models is that computation is compressed to just the effectual component on the dedicated cluster. On the other hand, ineffectual instructions must now wait longer for values from effectual instructions, since the effectual component is not replicated on all clusters.

Figure 4-5 shows how this heuristic works for a quad-cluster configuration. The dynamic instruction stream is broken into two components, predicted-effectual and

predicted-ineffectual instructions. All predicted-effectual instructions are steered to a single cluster, i.e., cluster I. This cluster, which is designated solely for the purpose of executing effectual instructions, is called the *effectual cluster*. All ineffectual instructions are steered to clusters other than the effectual cluster, i.e., clusters II, III, and IV, on the basis of dependence-based steering.

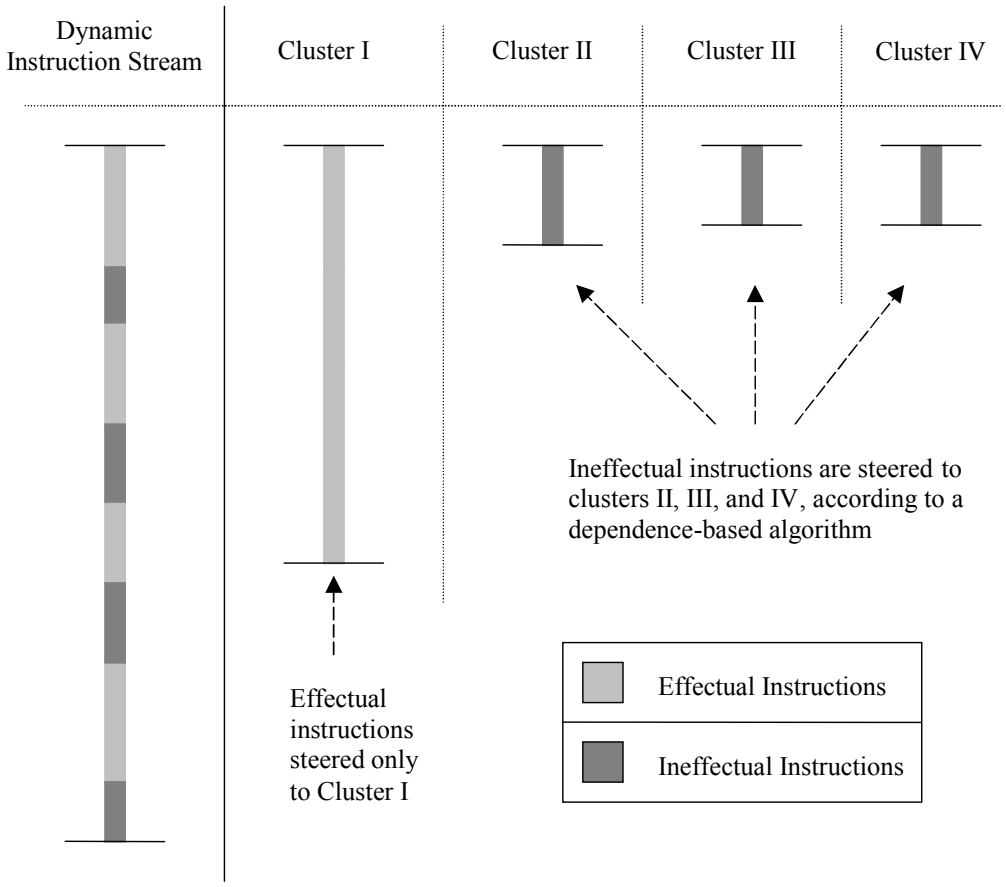

Figure 4-5: Dedicated cluster for effectual component.

Since there is no replication, no additional changes to the clustered microarchitecture are needed other than basing steering on effectualness.

4.4 Dedicated Cluster for Effectual Component with Store Distribution (DECS)

Depending on the type of program that is being run, the IR-predictor generally predicts about 10 – 65 % of the dynamic instruction stream to be ineffectual. A significant number of instructions are therefore predicted to be effectual. In the steering algorithm described in Section 4.3, all predicted-effectual instructions are executed on the effectual cluster, while all predicted-ineffectual instructions are executed on clusters other than the effectual cluster. Therefore, many instructions are executed on the effectual cluster and a comparatively lesser number of instructions are executed on each of the other clusters. Off-loading some of the effectual instructions to other clusters can ease pressure on the effectual cluster, but we must be careful that the resulting effectual inter-cluster communication is tolerable.

Predicted-effectual stores can be off-loaded to other clusters without introducing inter-cluster register communication. The only instructions that directly depend on stores are loads. In many cases, a dependent load consumes a store value long after it has been committed. Also, if a store misses in the cache, a dependent load has to wait for at least the time it takes to service the cache miss. If a store and a dependent load were steered to different clusters, the extra time for which the load has to wait due to inter-cluster communication latency is small in comparison to the time it takes to service a (L2) cache miss. Therefore, store-load dependences are relatively more tolerant of inter-cluster communication latency. There is still the issue of quickly resolving ambiguous and false store-load dependences (non-dependent effectual loads wait for the effectual store

address from other clusters), but as mentioned earlier, aggressive memory dependence speculation can very accurately remove false dependences [8][16].

For the algorithm described in this section, all store instructions are executed on clusters other than the effectual cluster. Thus, predicted-effectual instructions minus predicted-effectual stores are steered to the effectual cluster, while predicted-ineffectual instructions plus predicted-effectual stores are steered to clusters other than the effectual cluster.

Figure 4-6 shows how this algorithm works for a quad-cluster configuration. The dynamic instruction stream is represented as a collection of stores (predicted-effectual and predicted-ineffectual), predicted-effectual instructions (except predicted-effectual stores), and predicted-ineffectual instructions (except predicted-ineffectual stores). All stores and predicted-ineffectual instructions are steered to clusters II, III, and IV according to the DepR scheme. Predicted-effectual instructions minus predicted-effectual stores are steered to cluster I.

Again since there is no replication, there are no additional changes to the clustered microarchitecture other than basing steering on effectualness.

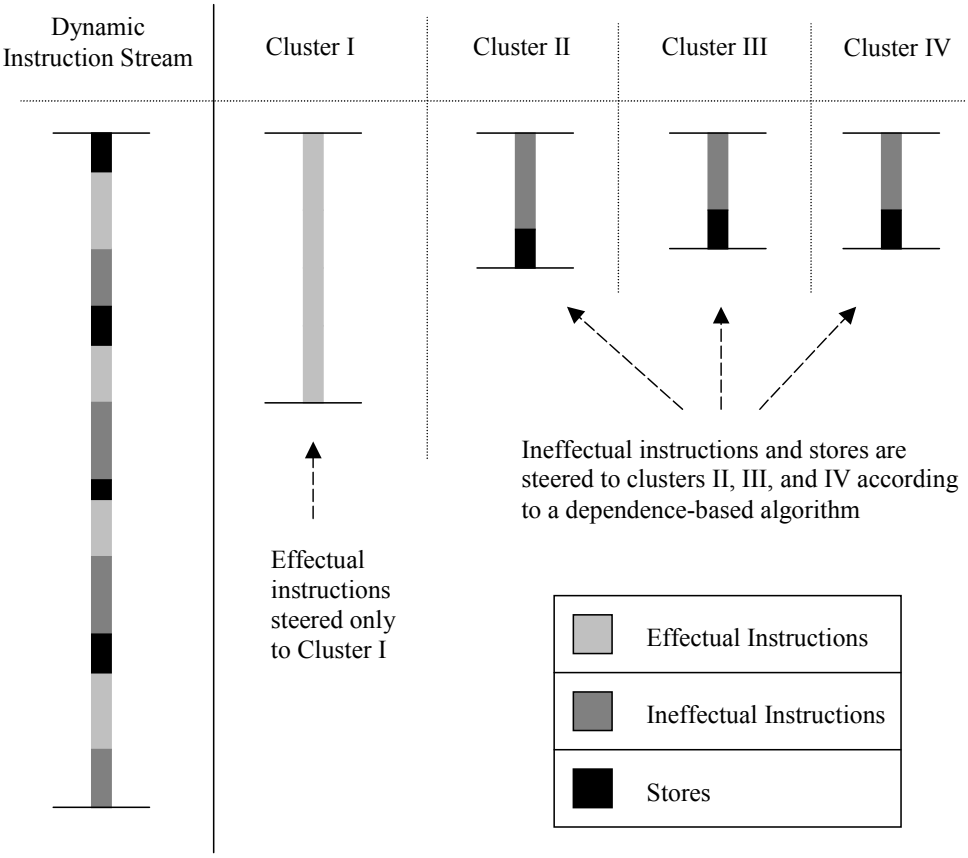

Figure 4-6: Dedicated cluster for effectual component with store distribution.

5 Simulation Methodology

5.1 Microarchitecture Configuration

A detailed cycle-accurate simulator forms the basis of the simulation environment. The Simplescalar ISA (PISA) [5] is used. The simulator models a dynamically scheduled processor with a seven-stage pipeline. There are separate L1 instruction and data caches. A unified L2 cache is used. A large IR-predictor is used for predicting the effectualness of instructions and predicting branch outcomes. An IR-detector based on implicit back-propagation is used [14]. Experiments are conducted for various non-clustered microarchitectures and equivalent clustered microarchitectures (i.e. same aggregate issue bandwidth and window size). For all non-clustered configurations, the issue queue and reorder buffer each have 256 entries and the load/store queue has 128 entries. Two different non-clustered configurations have been implemented, as follows.

- *Base_8*: The fetch, dispatch, issue, and retire bandwidths are 8 per cycle. There are 8 cache ports and 8 universal function units.
- *Base_12*: The fetch, dispatch, issue, and retire bandwidths are 12 per cycle. There are 12 cache ports and 12 universal function units.

The equivalent clustered microarchitectures corresponding to Base_8 are as follows.

- *dual_2x4_b8*: A dual-cluster configuration with an issue queue of 128 per cluster, an issue bandwidth of 4 per cluster, 4 cache ports per cluster, and 4 universal function units per cluster.

- *quad_4x2_b8*: A quad-cluster configuration with an issue queue of 64 per cluster, an issue bandwidth of 2 per cluster, 2 cache ports per cluster, and 2 universal function units per cluster.

The equivalent clustered microarchitecture corresponding to Base_12 is as follows.

- *quad_4x3_b12*: A quad-cluster configuration with an issue queue of 64 per cluster, an issue bandwidth of 3 per cluster, 3 cache ports per cluster, and 3 universal function units per cluster.

The various microarchitecture configurations of the system are summarized in Table 5-1.

Table 5-1: Microarchitecture configuration.

Microarchitecture			
	Monolithic Configuration	**Dual-Cluster Configuration**	**Quad-Cluster Configuration**
Reorder Buffer	256 entries		
Fetch / Dispatch / Retire Bandwidth	8, 12 per cycle		
Issue Queue	256 entries	128 entries per cluster	64 entries per cluster
Issue Bandwidth	8, 12 per cycle	4 per cluster per cycle	2, 3 per cluster per cycle
Function Units	8, 12 universal	4 universal per cluster	2, 3 universal per cluster
Cache Ports	8, 12	4 per cluster	2, 3 per cluster
Load / Store Queue	128 entries	128 entries per cluster	128 entries per cluster
Slipstream Components			
IR-Predictor	2^{20} entries, gshare-indexed		
	16 confidence counters per entry		
	confidence threshold = 3, 15, 31 …(varied)		
IR-Detector	implicit back-propagation, number of instructions buffered = 256		
Memory Hierarchy			
L1 I-Cache	64 KB, 4-way set-associative, 64-byte line size		
L1 D-Cache	64 KB, 4-way set-associative, 64-byte line size		
L2 Cache	512 KB unified instruction/data, 4-way set-associative, 64-byte line size		
Memory Access Times	L1 instruction hit = 1 cycle		
	L1 data hit = 1 cycle		
	L2 hit = 10 cycles		
	L2 miss = 70 cycles		

5.2 Benchmarks

Nine of the SPEC2000 integer benchmarks are used for the simulations. (Not all SPEC2000 benchmarks are used due to the volume of simulation runs.) The benchmarks are compiled with –O3 optimization using the Simplescalar compiler [5]. The first billion instructions are skipped and then 100 million instructions are simulated. The benchmarks and their input datasets are given in Table 5-2.

Table 5-2: Benchmarks and input data sets.

Benchmark	Input dataset
bzip	input.program 58
gap	-1./ -q –m 8M ref.in
gcc	expr.i –o expr.s (-O3 is hardwired)
gzip	input.program 16
parser	2.1.dict –batch
perl	-I./lib splitmail.pl 850 5 19 18 1500
twolf	ref
vortex	bendian1.raw
vpr	net.in arch.in place.out dum.out –nodisp –place_only –init_t5 –exit_t 0.005 –alpha_t 0.9412 –inner_num 2

6 Experimental Results

In this chapter, we study the sensitivity of conventional steering and slipstream-based steering to inter-cluster communication latency for the three clustered microarchitecture configurations described earlier. The inter-cluster communication latency is varied from 2 cycles to 16 cycles. The performance metric used is IPC relative to the corresponding non-clustered equivalent. That is, the performance of a clustered microarchitecture using various inter-cluster communication latencies is compared to an equivalent non-clustered microarchitecture that does not experience any inter-cluster communication penalty. For reference, we also show the relative performance of three small-scale superscalar processors, a 4-issue (4 way), 3-issue (3 way), and 2-issue (2 way) processor. The reason is that in some cases, inter-cluster communication latency may be high enough that simply using a narrower processor (4 way, 3 way, 2 way) is best. Table 6-1 shows the IPCs for 8-issue and 12-issue non-clustered processors.

Table 6-1: IPC for 8-issue and 12-issue non-clustered processors.

Benchmark	Instructions per Cycle (IPC)	
	Base_8	Base_12
bzip	6.67	7.98
gap	3.32	3.45
gcc	3.52	3.92
gzip	2.66	2.8
parser	1.79	1.87
perl	4.00	4.21
twolf	1.39	1.42
vortex	5.37	6.09
vpr	1.76	1.83

Note that Modulo_n steering heuristic is implemented for all benchmarks with n equal to 3. The conventional steering heuristics, Modulo_3 (Mod3), Load-Cut (LC),

Branch-Cut (BC), and Least-Loaded (LL), are collectively called *distribution-based* steering, while Dependence-Based (Dep0) and Readiness-Dependence-Based (DepR) are collectively called *dependence-based* steering. The steering algorithms proposed in this thesis are collectively called *slipstream-based* steering.

Experiments are carried out using the three clustered microarchitecture configurations described in Section 5. The results are organized as follows. In the first section, the performance of conventional steering is studied for all three clustered microarchitecture configurations. This enables us to narrow down conventional steering heuristics that are best for comparison with slipstream-based steering. We then study the performance of slipstream-based steering and compare it with the best of conventional steering for all the configurations.

Graphs are plotted with IPC relative to the equivalent non-clustered microarchitecture on the y-axis and inter-cluster communication latency (in cycles) on the x-axis.

6.1 Conventional steering

Results for conventional steering are presented for only five of the SPEC2000 benchmarks. These benchmarks are gap, gcc, perl, twolf, and vortex. These benchmarks represent the entire spectrum of IPC performance of the SPEC2000 benchmark suite.

6.1.1 Dual_2x4_b8

Figure 6-1 through Figure 6-5 show the relative performance of conventional steering (distribution-based and dependence-based) on dual_2x4_b8 (dual-cluster, aggregate issue width of 8), with respect to Base_8, for the five benchmarks. The first observation is that the performance of all steering heuristics degrades with an increase in inter-cluster communication latency. The performance of distribution-based steering does not compare well with dependence-based steering for the entire latency range and the relative performance degradation of distribution-based steering (BC, LC, Mod3, and LL) is far greater than that of dependence-based steering (Dep0 and DepR). Dependence-based algorithms do well at all latencies because they take both load balancing and dependence information into account while steering. At lower inter-cluster latencies, cluster issue bandwidth constrains performance more than inter-cluster communication. Therefore, the performance of some of the distribution-based algorithms comes close to that of the dependence-based algorithms. However, at higher latencies, the penalty for inter-cluster communication is the larger problem, and distribution-based algorithms do not explicitly address it.

The general trend between the two dependence-based algorithms is that DepR is more latency-tolerant than Dep0, for all benchmarks except perl, i.e., Dep0 has a steeper gradient of performance degradation than DepR. At lower latencies, Dep0 and DepR have nearly the same relative performance. However, at higher latencies, DepR performs distinctly better as it takes timing information into account while steering.

For *perl* (see Figure 6-3), in contrast to the other benchmarks, Dep0 outperforms DepR for all inter-cluster latencies. Also, the performance of DepR degrades more with increasing latency than Dep0. This is because the average time spent by a load in disambiguation is greater for the DepR scheme than the Dep0 scheme (see Figure 6-6). Apparently, Dep0 steers store-load dependences more efficiently than DepR, even though neither of the algorithms specifically consider memory dependences.

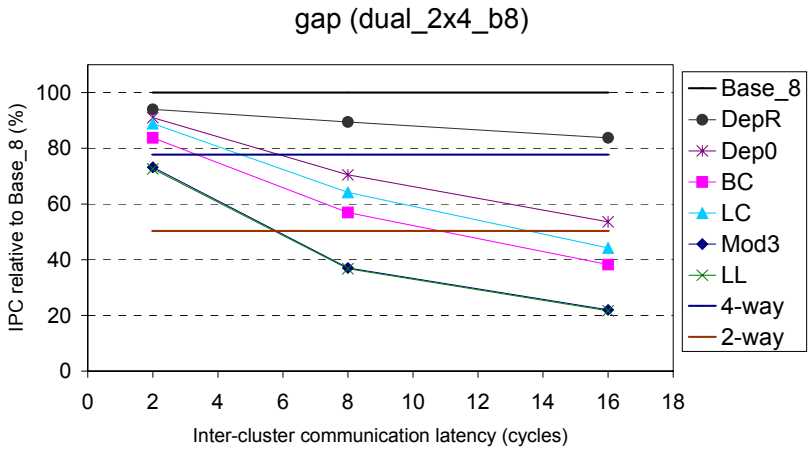

Figure 6-1: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, for *gap*.

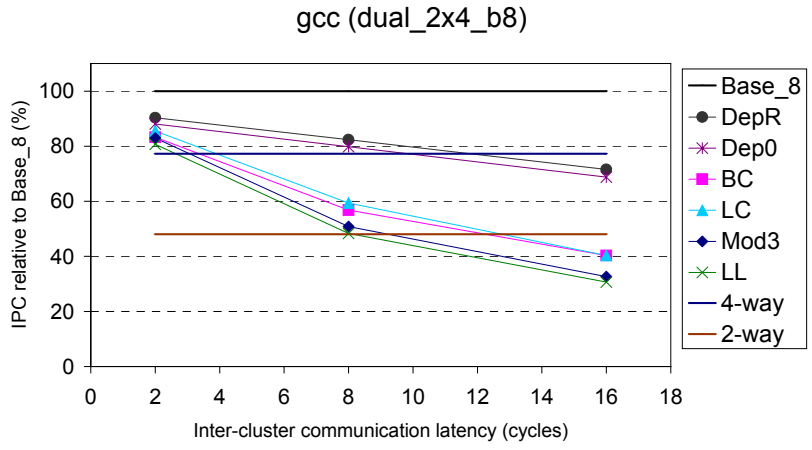

Figure 6-2: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, for *gcc*.

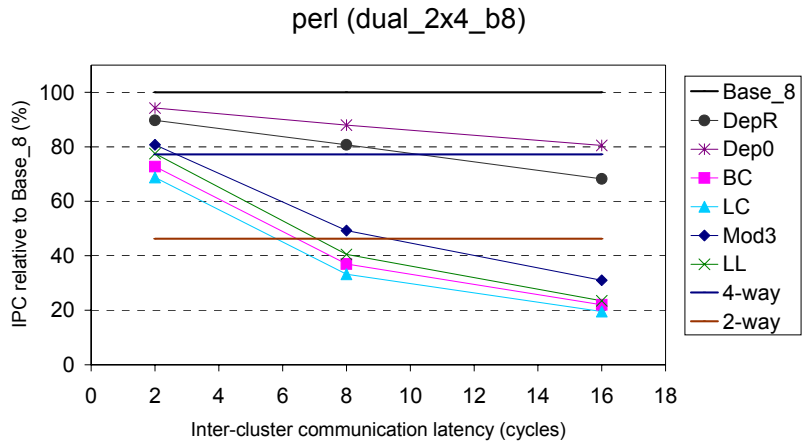

Figure 6-3: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, for *perl.*

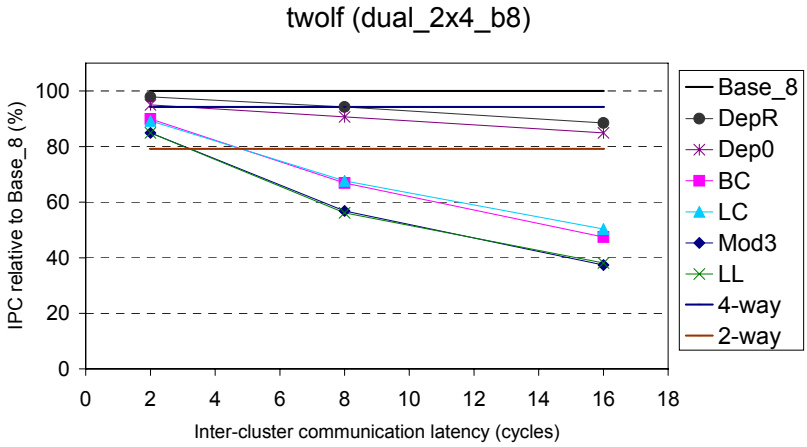

Figure 6-4: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, for *twolf.*

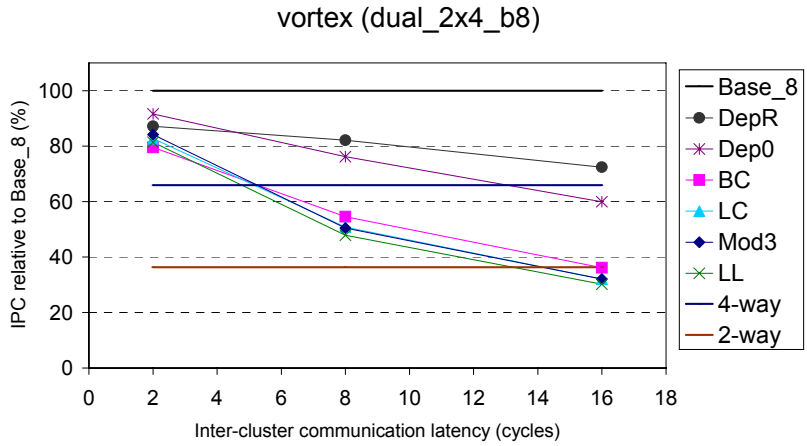

Figure 6-5: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, for *vortex.*

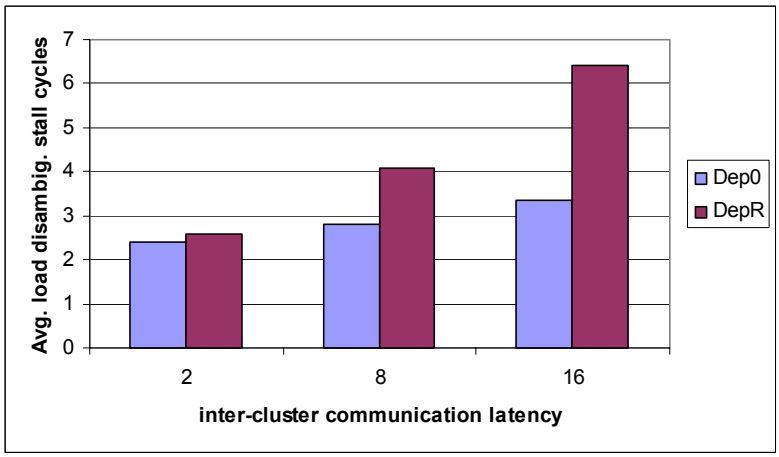

Figure 6-6: Average load disambiguation stall cycles of Dep0 and DepR on dual_2x4_b8, for *perl*.

Figure 6-7 shows the relative performance of conventional steering on dual_2x4_b8, with respect to Base_8, averaged across the five benchmarks. The trend observed in a majority of the benchmarks is confirmed. Dep0 and DepR outperform distribution-based steering at all latencies, and DepR is more latency tolerant than Dep0. At 2-cycle latency, the performance of Dep0 is comparable to that of DepR: both have a relative performance of 92%. On the other hand, relative performance of Branch-Cut (BC) is only 82% at 2-cycle latency. However, at 16-cycle latency, DepR has a relative performance of 77% compared to 70% for Dep0, while the relative performance of BC is only 35%. Clearly, dependence-based steering outperforms distribution-based steering. Thus, slipstream-based steering will only be compared to dependence-based steering.

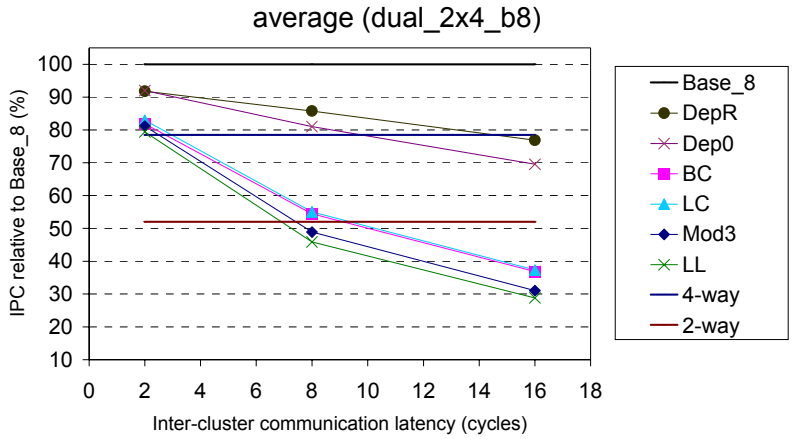

Figure 6-7: Relative performance of conventional steering on dual_2x4_b8, with respect to Base_8. Results are averaged across five benchmarks.

6.1.2 Quad_4x2_b8

Figure 6-8 through Figure 6-12 show the relative performance of conventional steering on quad_4x2_b8 (quad-cluster, aggregate issue width of 8), with respect to Base_8, for the five benchmarks. Some of the trends observed for dual_2x4_b8 are observed here. Dependence-based steering outperforms distribution-based steering, both in terms of relative performance and latency-tolerance. This can be seen in Figure 6-13, which shows the relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, averaged across all benchmarks. When latency increases from 2 to 16 cycles, the relative performance of DepR drops from 81% to 62%, while that of BC drops from 71% to 29%.

Among the dependence-based algorithms, neither of the two heuristics has a distinct advantage over the other. At 2-cycle latency, DepR performs better for gap, gcc, perl, and twolf, while at 16-cycle latency, Dep0 performs better for gcc, perl, and vortex. Figure 6-13 shows that the relative performance of Dep0 and DepR, averaged across all

benchmarks, are nearly the same. This happens because the execution bandwidth on each individual cluster is quite low (2 per cluster), and that is what really constrains performance.

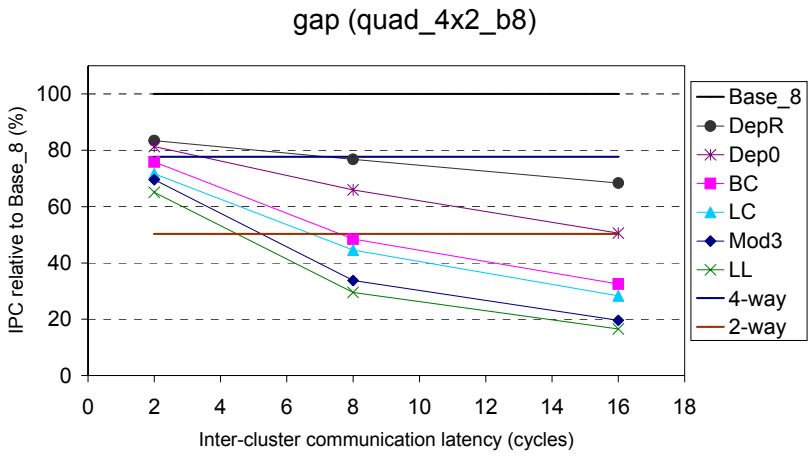

Figure 6-8: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, for *gap*.

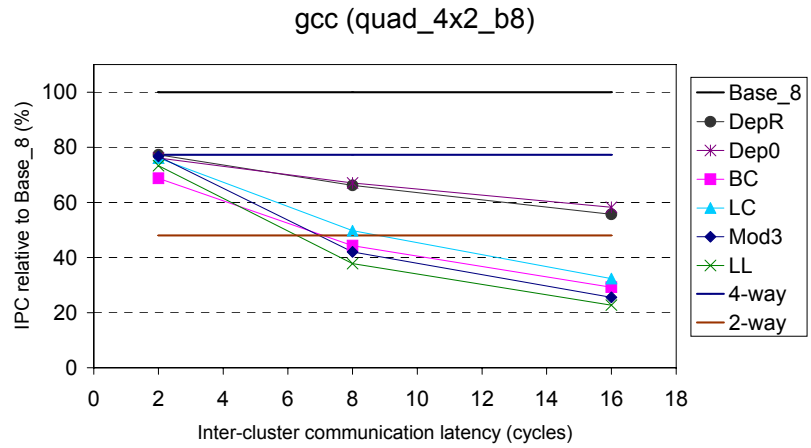

Figure 6-9: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, for *gcc*.

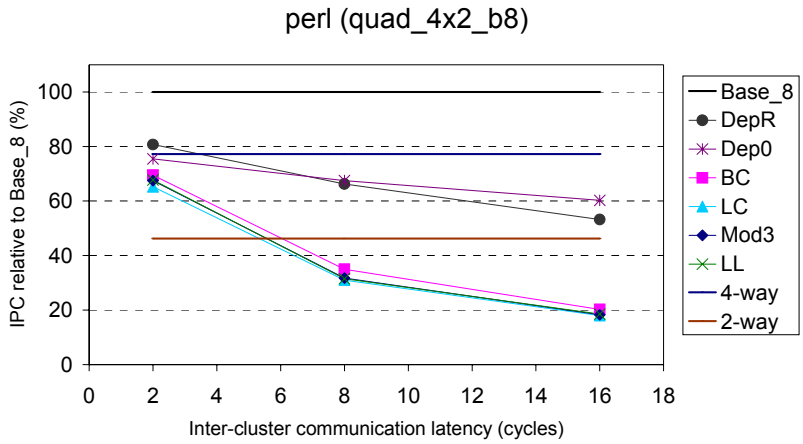

Figure 6-10: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, for *perl.*

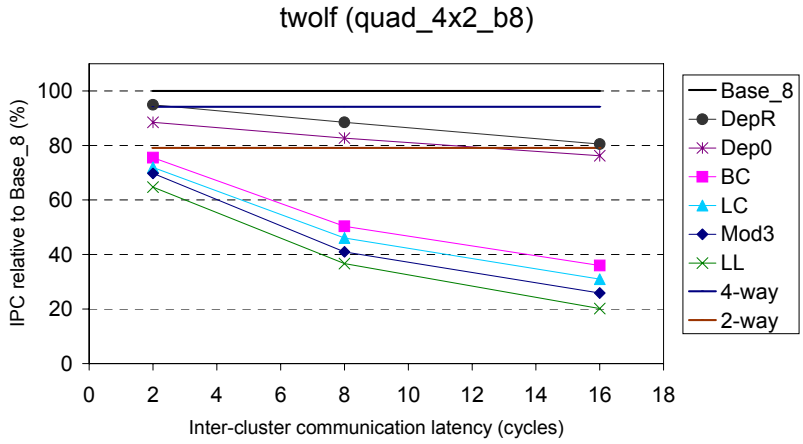

Figure 6-11: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, for *twolf.*

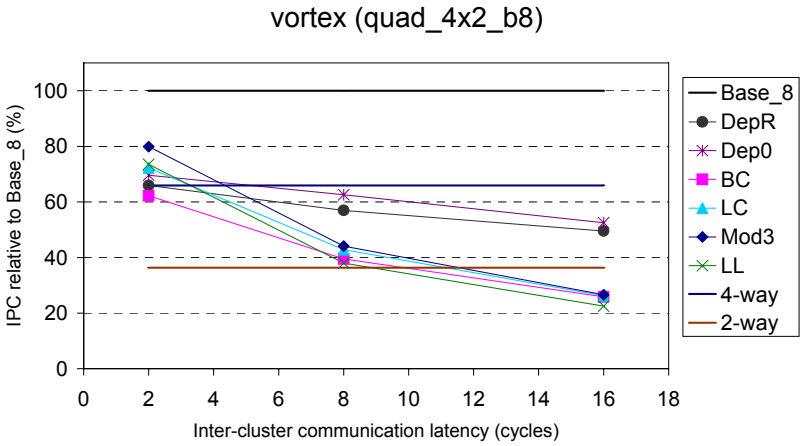

Figure 6-12: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8, for *vortex.*

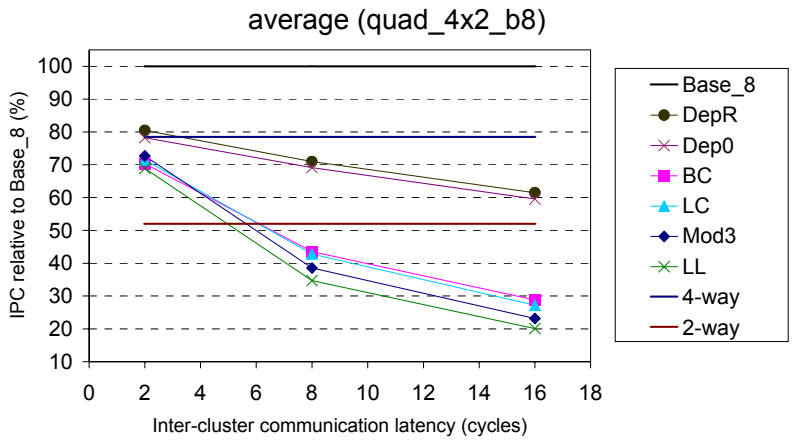

Figure 6-13: Relative performance of conventional steering on quad_4x2_b8, with respect to Base_8. Results are averaged across five benchmarks.

6.1.3 Quad_4x3_b12

Figure 6-14 through Figure 6-18 show the relative performance of conventional steering on quad_4x3_b12 (quad-cluster, aggregate issue width of 12), with respect to Base_12, for the five benchmarks. Similar trends observed for dual_2x4_b8 are observed here. Dependence-based steering outperforms distribution-based steering. This can be seen in Figure 6-19, which shows the relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, averaged across all benchmarks. The relative performance of DepR drops from 86% to 63% for the latency range, while that of BC drops from 75% to 30%. Between the two dependence-based schemes, DepR outperforms Dep0 for all benchmarks except perl. For perl (see Figure 6-16), Dep0 outperforms DepR for reasons mentioned earlier (fewer load disambiguation stalls).

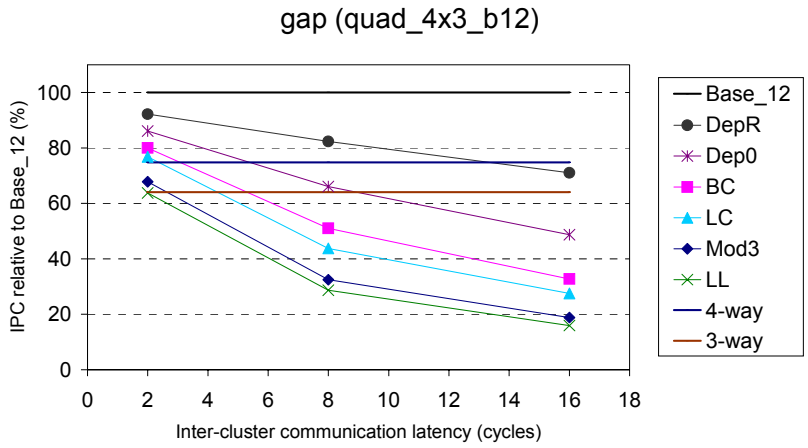

Figure 6-14: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, for *gap*.

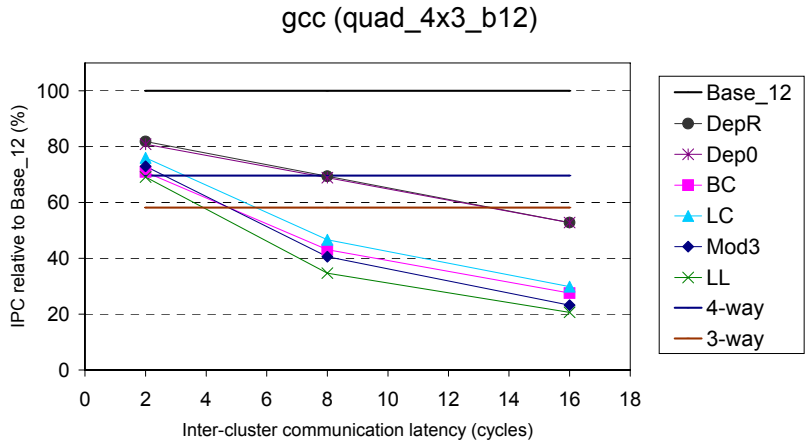

Figure 6-15: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, for *gcc*.

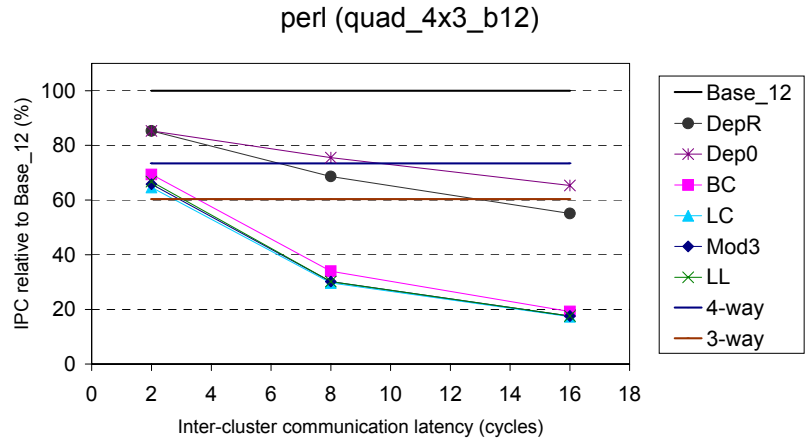

Figure 6-16: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, for *perl*.

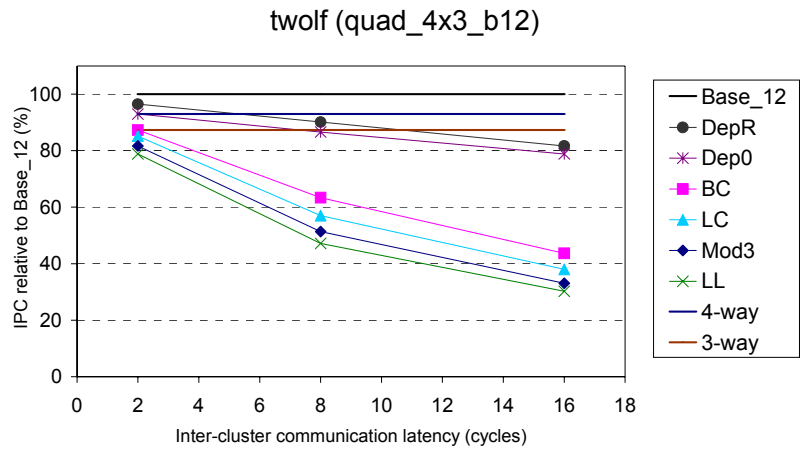

Figure 6-17: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, for *twolf*.

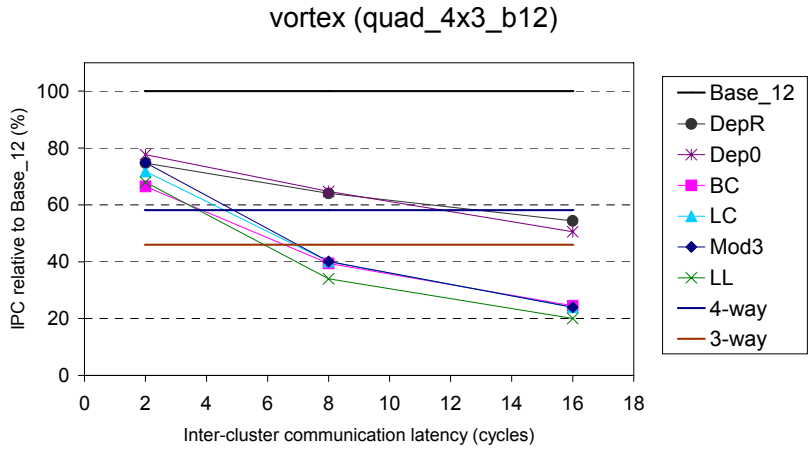

Figure 6-18: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12, for *vortex*.

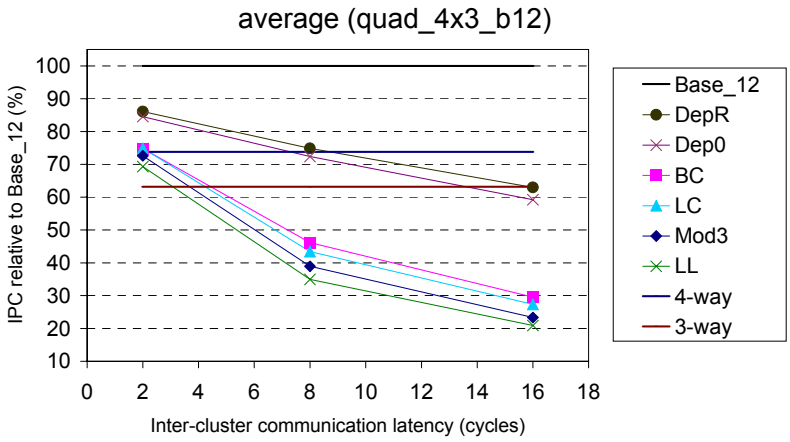

Figure 6-19: Relative performance of conventional steering on quad_4x3_b12, with respect to Base_12. Results are averaged across five benchmarks.

6.2 Slipstream-based steering

Rep0 and RepS are collectively called replication-based algorithms and DEC0 and DECS are collectively called dedicated-cluster-based algorithms. Note that for slipstream-based steering, parameters such as instruction-removal threshold and instruction-removal criteria are varied.

For an instruction to be predicted ineffectual by the IR-predictor, the confidence counter associated with the instruction must get saturated. The time it takes to saturate the counter depends on the confidence threshold. Therefore, the percentage of instructions predicted to be ineffectual (or effectual) would differ with different thresholds. Confidence thresholds of 3 and 15 are used in this thesis. With lower confidence thresholds, the number of predicted-ineffectual instructions will increase, but it will be accompanied by a corresponding increase in IR-mispredictions. Thus, a performance trade-off has to be considered between reducing demand for cluster issue bandwidth and increasing the IR-misprediction rate and therefore inter-cluster communication among effectual instructions.

Instruction-removal criteria determine what kinds of instructions are predicted effectual or ineffectual. The non-modifying-write criterion (also called Write Same Value (WSV)) is configurable in our simulator. If the WSV criterion is not used, then non-modifying writes will not be selected for removal. This typically leads to fewer predicted-ineffectual instructions. This criterion has special significance in the context of slipstream-based steering. With WSV as an instruction-removal criterion, some non-

modifying writes may be predicted ineffectual. These instructions will be distributed among the clusters in the replication-based schemes and will not be steered to the effectual cluster in the dedicated-cluster-based schemes. Thus, effectual instructions that depend on these instructions are exposed to inter-cluster communication latency. We expect that algorithms that use WSV as an instruction-removal criterion will not be highly latency-tolerant because they increase inter-cluster communication between effectual instructions.

The crux of the problem with WSV is that a form of value speculation is needed to exploit it in our microarchitecture. A predicted-ineffectual instruction classified as WSV implies that the value currently available in the register file (or memory location) is most likely equal to the value that will be produced. Thus, to exploit the WSV ineffectual criterion, effectual dependent instructions need to issue speculatively with the old register file value. We leave this aspect for future work.

The number of predicted-ineffectual instructions as a percentage of dynamic instructions is shown in Figure 6-20 for five benchmarks. Results are presented for confidence thresholds of 3 and 15, with and without the WSV criterion. In the graph, benchmarks are shown on the x-axis and the % predicted-ineffectual instructions are shown on the y-axis. C3 (WSV) indicates a confidence threshold of 3 and the WSV criterion are used, while C3 indicates that the WSV criterion is not used. The percentage of predicted-ineffectual instructions increases with a decrease in the confidence threshold, as expected. More instructions are predicted ineffectual when the WSV policy

is used. The only exception is *twolf*, for a confidence threshold of 15: more instructions are removed without WSV than with WSV.

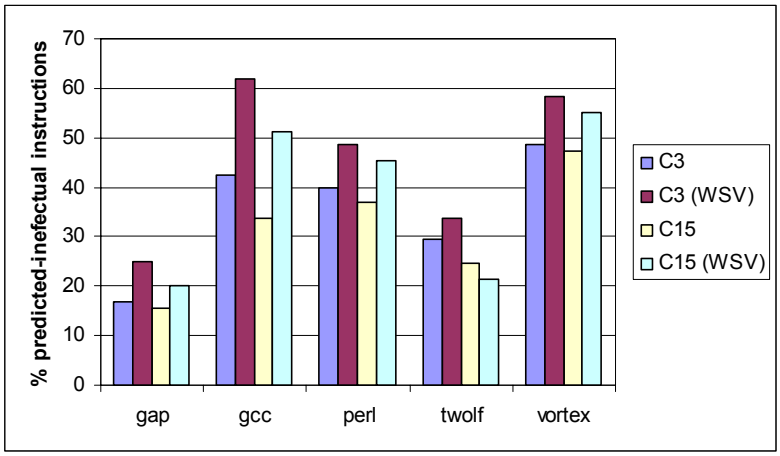

Figure 6-20: Percentage of predicted-ineffectual instructions.

A general trend is that for a given slipstream-based steering heuristic, lowering the confidence threshold from 15 to 3 reduces the latency-tolerance of the heuristic. With a lower confidence threshold, more instructions are predicted to be ineffectual, but the number of IR-mispredictions increase. Easing of pressure on the cluster issue bandwidth improves performance at low latencies, but the increase in inter-cluster communication due to IR-mispredictions degrades performance at higher latencies. Therefore, results for a threshold of 15 are presented in this section, while results for a threshold of 3 are deferred to the appendix. Also, algorithms that do not use the WSV criterion are more latency tolerant than those that use WSV. The performance degradation for algorithms using WSV is very high. Therefore, results for the WSV criterion are also deferred to the appendix. From this point onwards, we only consider a confidence threshold of 15 and disable the WSV criterion.

6.2.1 Dual_2x4_b8

Figure 6-21 through Figure 6-29 show the relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and the best conventional steering (i.e., dependence-based steering) on dual_2x4_b8 (dual cluster, aggregate issue width of 8), with respect to Base_8, for all benchmarks. Figure 6-30 shows the relative performance of slipstream-based steering and dependence-based steering on dual_2x4_b8, with respect to Base_8, averaged across all benchmarks.

6.2.1.1 Trends among slipstream-based steering

Replication-based steering is more latency-tolerant than dedicated-cluster-based steering for all benchmarks. This was expected, because the replication model only has inter-cluster communication between ineffectual instructions, whereas the dedicated-cluster model additionally has inter-cluster communication between effectual and ineffectual instructions. From Figure 6-30, it can be seen that Rep0, RepS, DEC0, and DECS experience a relative performance drop of 1%, 5%, 4%, and 23%, respectively, on average, over the entire latency range. Thus, Rep0 is the most latency-tolerant among all slipstream-based steering heuristics.

By distributing stores, the replication-based schemes and the dedicated-cluster-based schemes become less latency-tolerant. At low latencies, the advantage of executing fewer instructions on a cluster results in better performance, especially in the case of RepS. However, inter-cluster communication between effectual stores and loads significantly degrades performance at higher latencies, especially in the case of DECS.

This is the reason that DECS has the steepest gradient. Figure 6-30 shows that the relative performance of Rep0 drops from 88% to 87% over the latency range, while that of RepS drops from 90% to 86%, and that of DECS drops from 92% to 67%.

DEC0 outperforms all other slipstream-based steering heuristics for all benchmarks except gap and vpr. On average, the relative performance of DEC0 drops from 94% to 90% across the latency range (see Figure 6-30). The percentage of predicted-ineffectual instructions is only 16% for gap and 21% for vpr. In both cases, DECS has the best performance at low latencies. DECS frees up execution resources, which are highly constrained due to the large number of instructions being executed on the effectual cluster. This provides a benefit at lower latencies, but performance degrades at higher latencies due to inter-cluster communication between effectual stores and loads. The performance of other algorithms does not degrade because their execution is primarily constrained by the cluster issue bandwidth and increase in inter-cluster communication latency has little effect on performance.

6.2.1.2 Comparison of slipstream-based steering with conventional steering

DEC0 outperforms dependence-based steering for all benchmarks except *gap*. For *gap*, which has a low number of predicted-ineffectual instructions, dependence-based steering does a better job of load balancing than slipstream-based steering. However, dependence-based steering is not very latency-tolerant (steep gradient), even though it has better performance than slipstream-based steering at all latencies.

On average, all slipstream-based steering heuristics except DECS are highly latency-tolerant and have better performance compared to dependence-based steering (see Figure 6-30). The relative performance of DepR drops from 92% to 71%, whereas relative performance of DEC0 drops from 94% to 90%, on average. On average, at 16-cycle latency, it is better to use only a single 4-issue cluster (relative performance of 4 way is 80%) than dual 4-issue clusters with dependence-based steering (relative performance is 71%). However, slipstream-based steering extends the usefulness of a clustered microarchitecture to very high latencies.

Figure 6-31 shows the IPC improvement of slipstream-based steering with respect to the best conventional steering, averaged across all benchmarks. At 2-cycle latency, DEC0 performs the same as dependence-based steering. However, Rep0, RepS, and DECS do not perform as well. At higher latencies, DEC0, Rep0, and RepS record a performance improvement over dependence-based steering. DEC0, Rep0, and RepS improve performance by 10%, 4%, and 5%, at 8-cycle latency, respectively, and by 24%, 18%, and 16%, at 16-cycle latency, respectively.

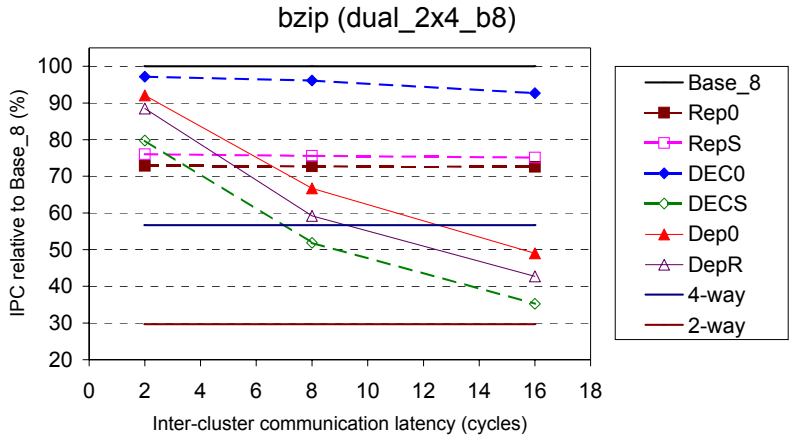

Figure 6-21: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *bzip*.

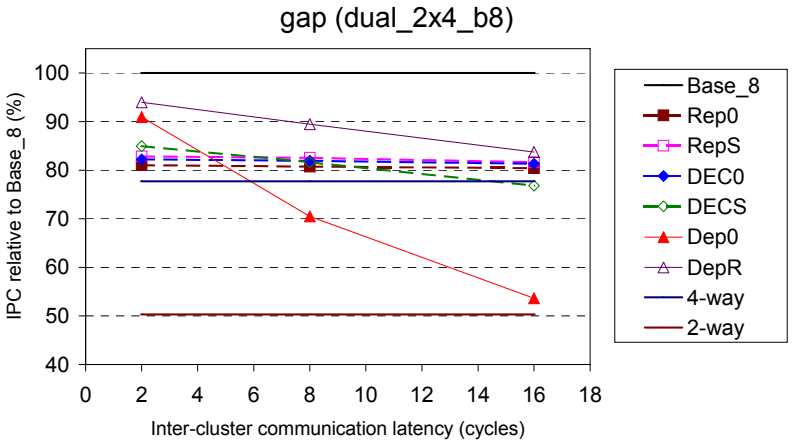

Figure 6-22: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *gap*.

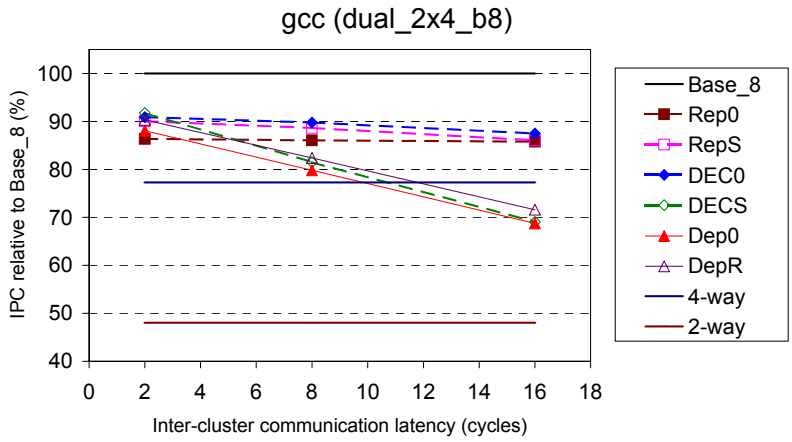

Figure 6-23: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *gcc*.

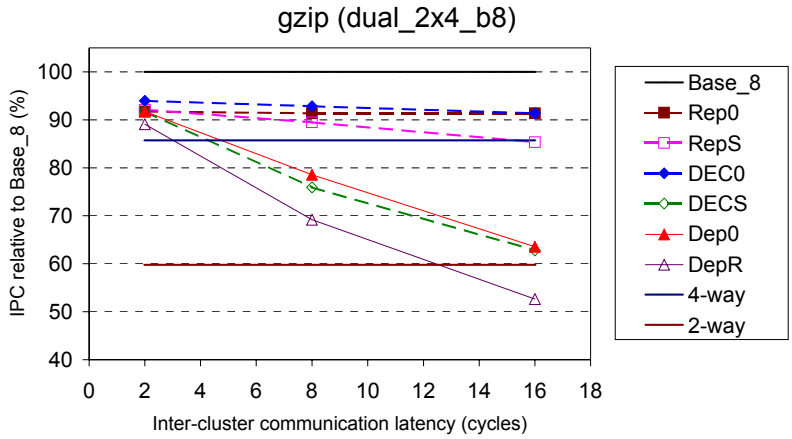

Figure 6-24: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *gzip*.

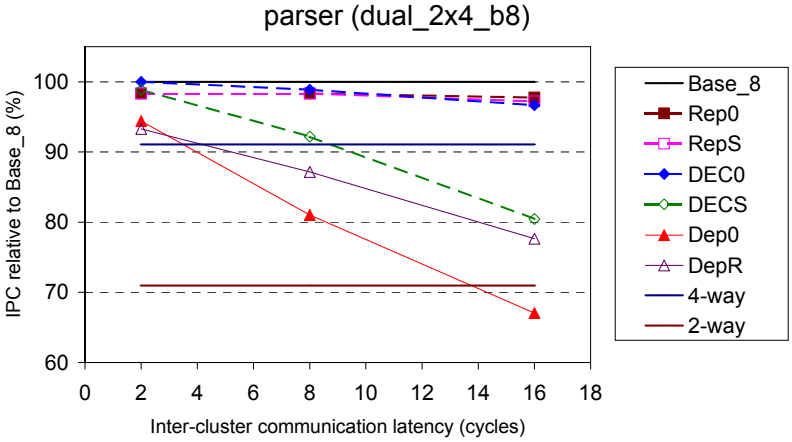

Figure 6-25: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *parser*.

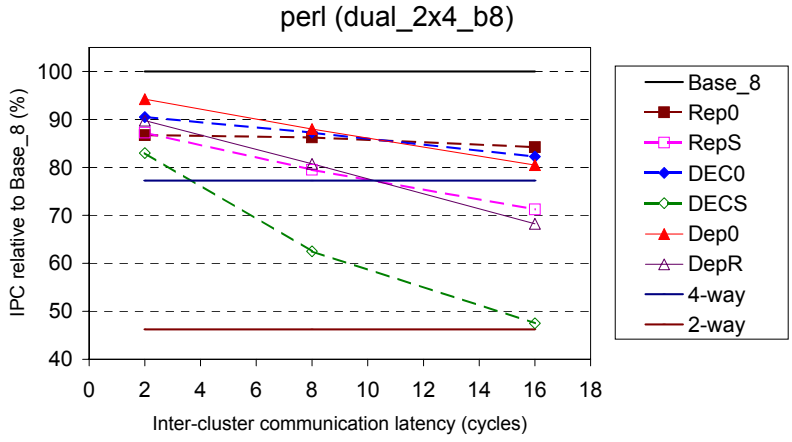

Figure 6-26: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *perl*.

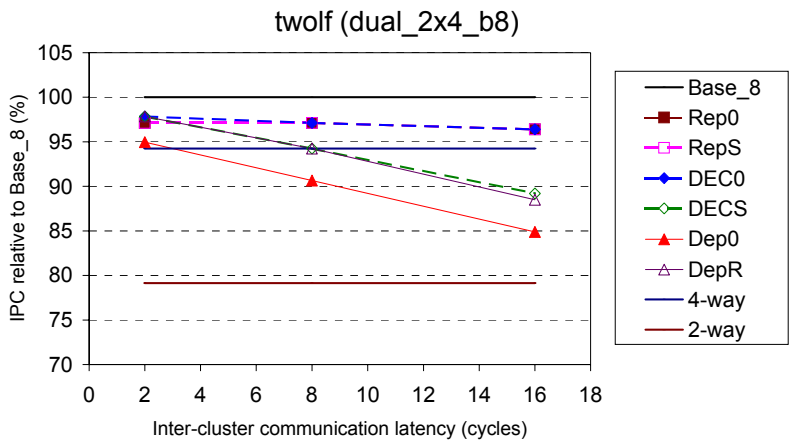

Figure 6-27: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *twolf*.

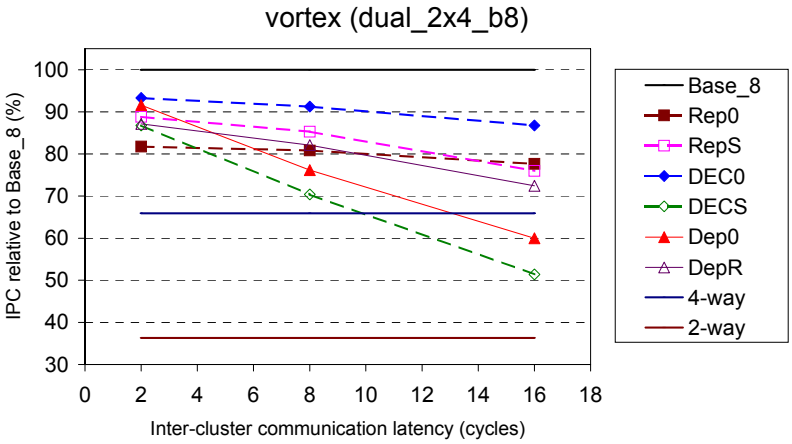

Figure 6-28: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *vortex*.

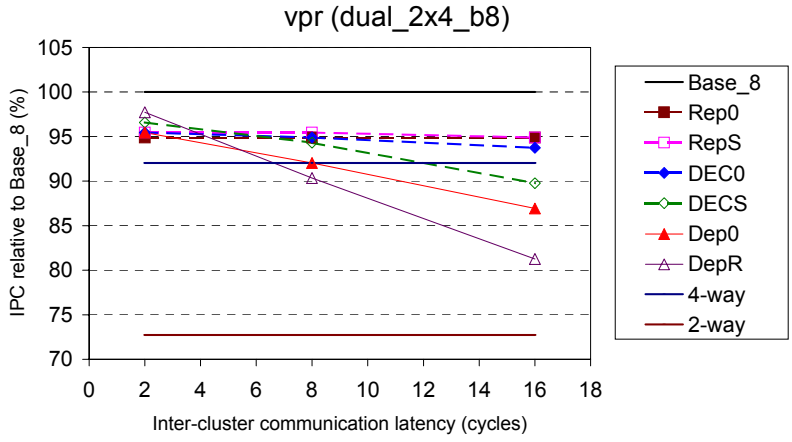

Figure 6-29: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8, for *vpr*.

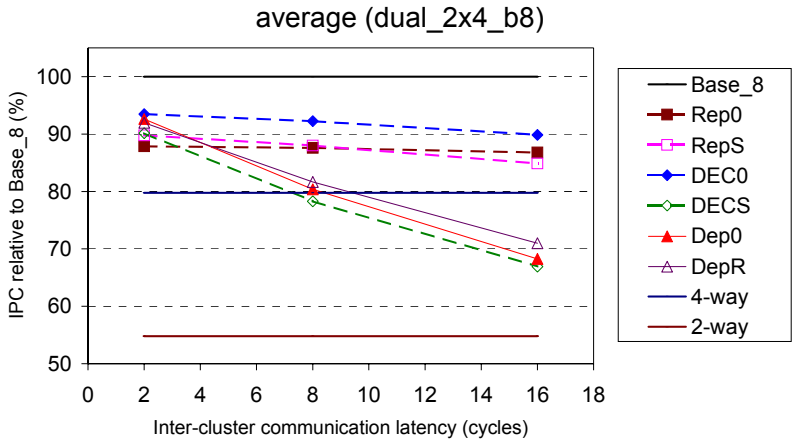

Figure 6-30: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on dual_2x4_b8, with respect to Base_8. Results are averaged across all benchmarks.

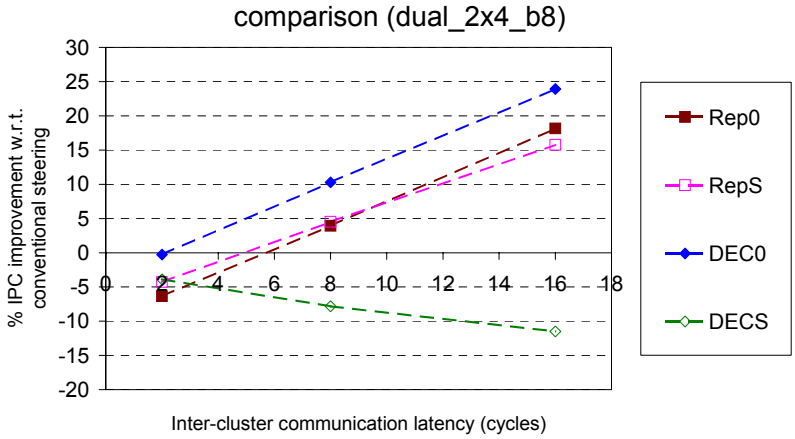

Figure 6-31: IPC improvement of slipstream-based steering (confidence threshold of 15 and no WSV) with respect to dependence-based steering on dual_2x4_b8. Results are averaged across all benchmarks.

6.2.2 Quad_4x2_b8

Figure 6-32 through Figure 6-40 show the relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and the best conventional steering (i.e., dependence-based steering) on quad_4x2_b8 (quad-cluster, aggregate issue width of 8), with respect to Base_8, for all benchmarks. Figure 6-41 shows the relative

performance of slipstream-based steering and dependence-based steering on quad_4x2_b8, with respect to Base_8, averaged across all benchmarks.

6.2.2.1 Trends among slipstream-based steering

DECS outperforms other slipstream-based steering schemes at a latency of 2 cycles for all benchmarks. At 2-cycle latency, on average, DECS has a relative performance of 77% compared to 71% for DEC0 and 72% for RepS (see Figure 6-41). DECS executes the least number of instructions per cluster, compared to the other slipstream-based steering heuristics. As the issue bandwidth per cluster is very low (2 per cycle), this gives DECS an edge over other slipstream-based steering heuristics at low latencies. However, DECS is not latency tolerant, for reasons mentioned earlier.

Rep0 underperforms DEC0 and RepS for all benchmarks. Pressure on the execution bandwidth of each cluster is highest for Rep0, because it executes the highest number of instructions on each cluster among all slipstream-based steering heuristics. In this configuration, performance is primarily constrained by the meager resources available within each cluster. Thus, even though Rep0 is highly latency-tolerant, DEC0 and RepS, which execute fewer instructions on each cluster, outperform it and have comparable latency-tolerance.

Another trend is that RepS outperforms all other steering schemes at the 16-cycle latency. At 16-cycle latency, on average, RepS has a relative performance of 70% compared to 66% for Rep0 and 69% for DEC0 (see Figure 6-41). Replication provides

latency-tolerance, while distribution of stores eases issue bandwidth pressure, and improves performance.

6.2.2.2 Comparison of slipstream-based steering with conventional steering

Slipstream-based steering (except for DECS) is highly latency-tolerant for this configuration, compared to dependence-based steering. DepR outperforms slipstream-based steering at 2-cycle latency for all benchmarks except parser. On the other hand, Rep0, RepS, and DEC0 outperform dependence-based steering at 16-cycle latency for all benchmarks except gap.

Figure 6-42 shows the IPC improvement of slipstream-based steering with respect to the best of the conventional steering methods, averaged across all benchmarks. DECS underperforms dependence-based steering at all latencies. For latencies less than 8-cycles, DEC0, Rep0, and RepS underperform dependence-based steering. However, at 16-cycle latency, DEC0, Rep0, and RepS improve IPC by 16%, 11%, and 17%, respectively.

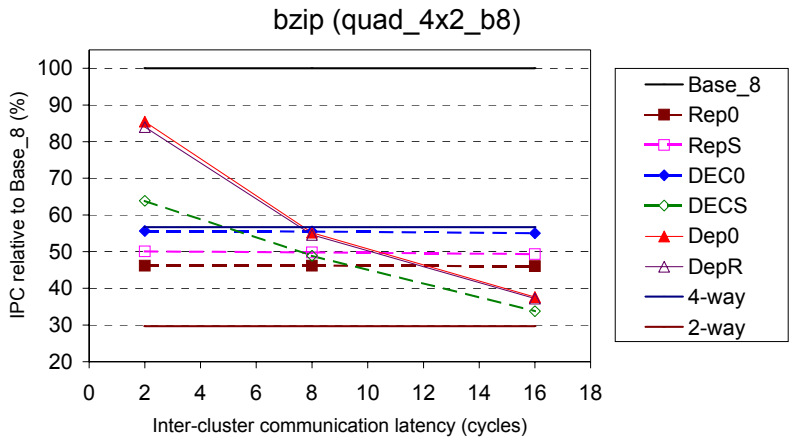

Figure 6-32: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *bzip*.

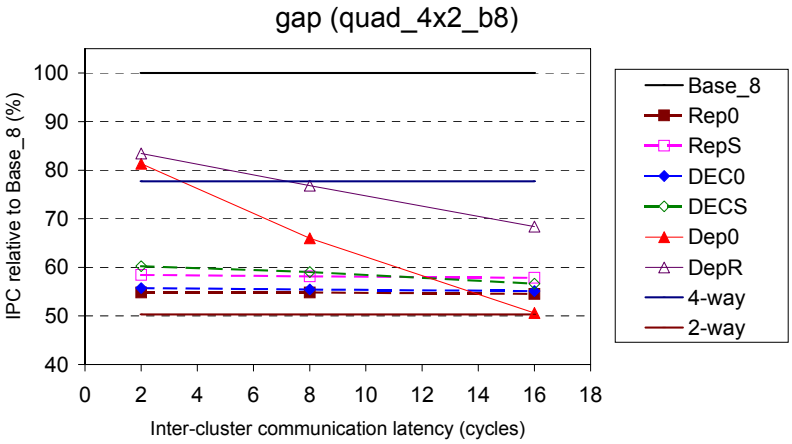

Figure 6-33: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *gap*.

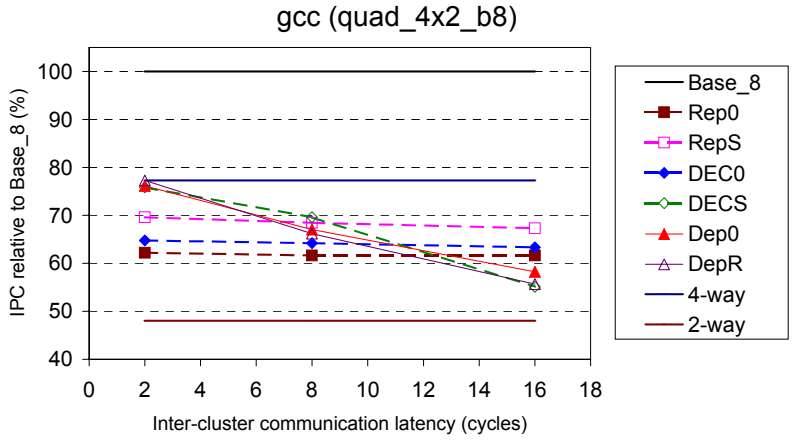

Figure 6-34: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *gcc*.

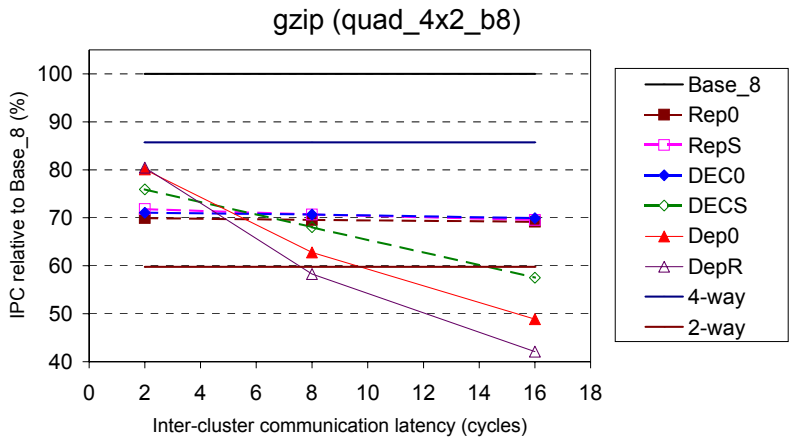

Figure 6-35: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *gzip*.

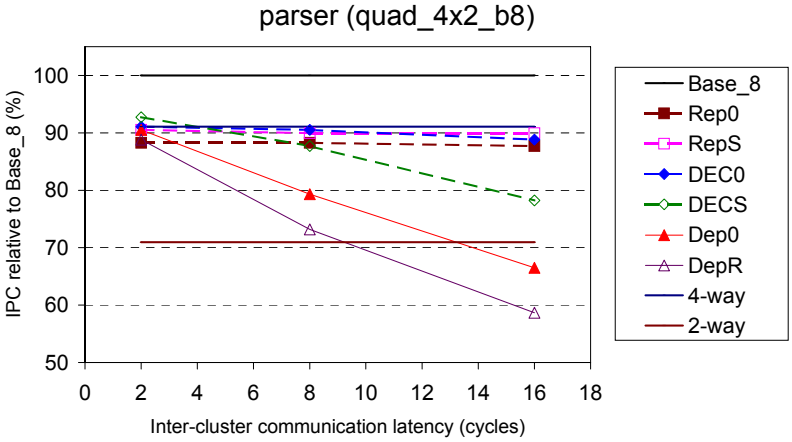

Figure 6-36: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *parser*.

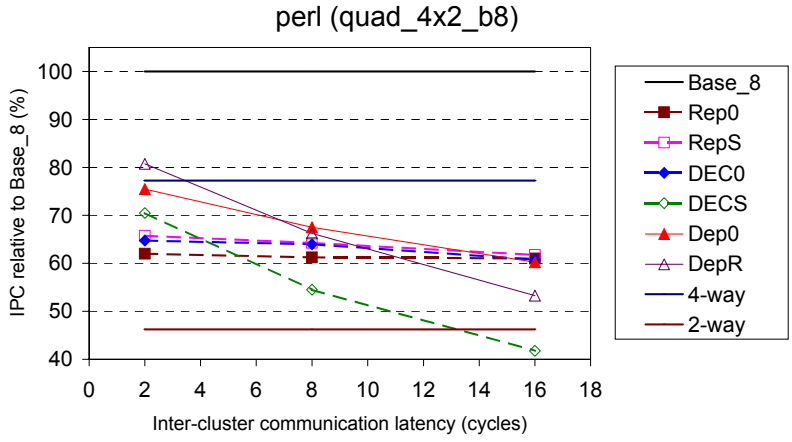

Figure 6-37: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *perl*.

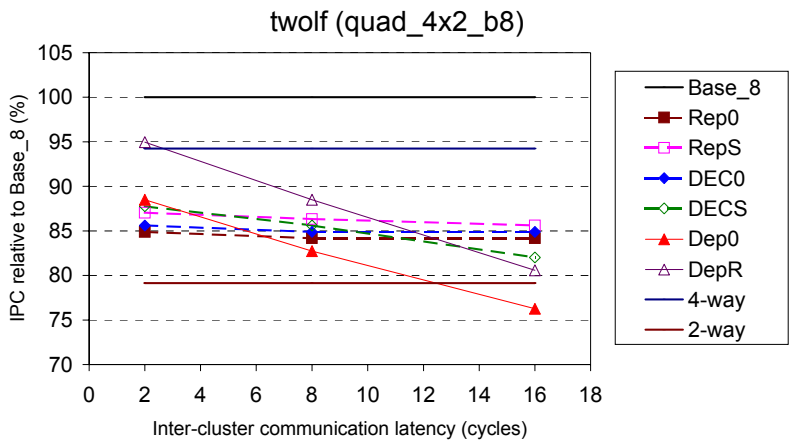

Figure 6-38: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *twolf*.

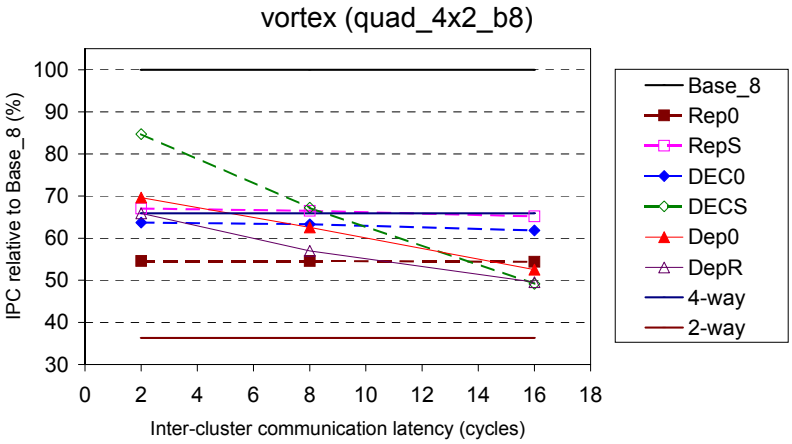

Figure 6-39: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *vortex*.

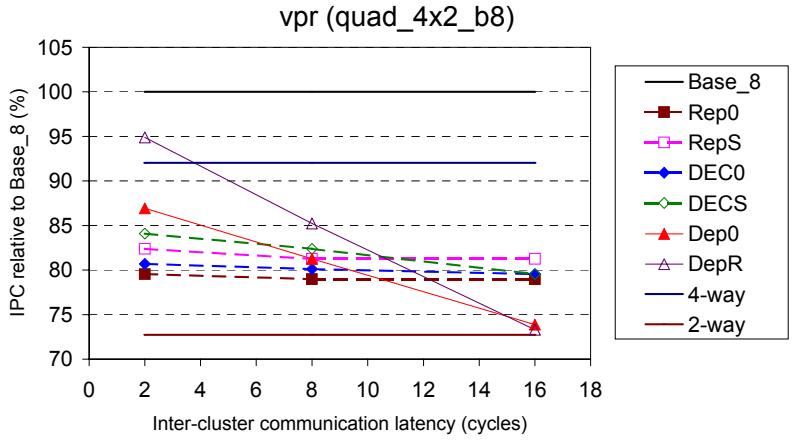

Figure 6-40: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8, for *vpr*.

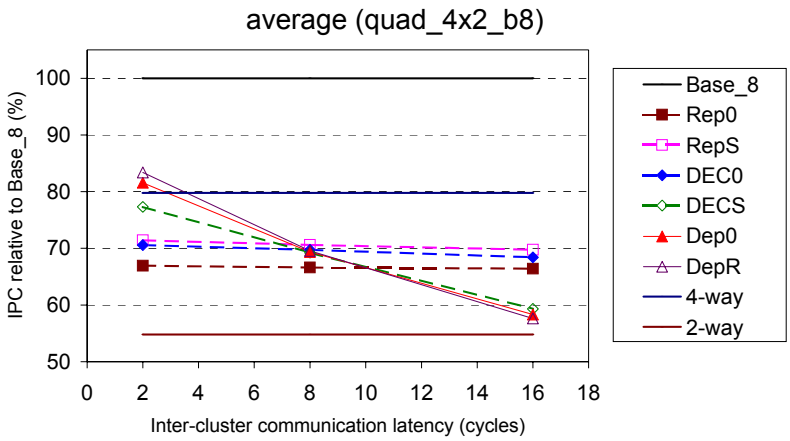

Figure 6-41: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x2_b8, with respect to Base_8. Results are averaged across all benchmarks.

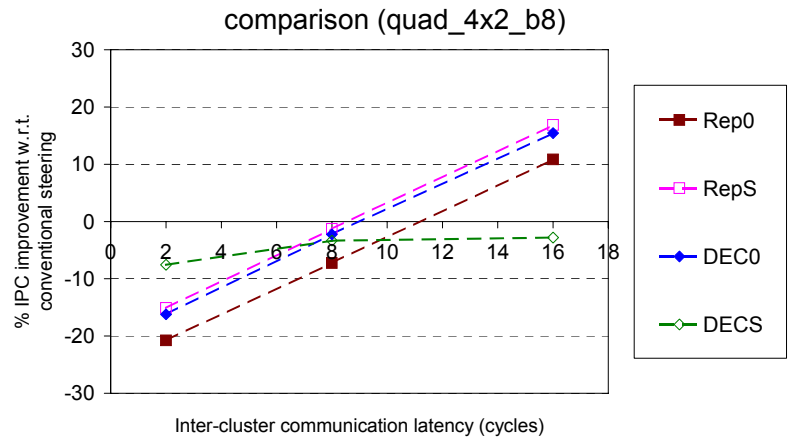

Figure 6-42: IPC improvement of slipstream-based steering (confidence threshold of 15 and no WSV) with respect to dependence-based steering on quad_4x2_b8. Results are averaged across all benchmarks.

6.2.3 Quad_4x3_b12

Figure 6-44 through Figure 6-52 show the relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and the best conventional steering (i.e., dependence-based steering) on quad_4x3_b12 (quad-cluster, aggregate issue width of 12), with respect to Base_12, for all benchmarks. Figure 6-53 shows the

relative performance of slipstream-based steering and dependence-based steering on quad_4x3_b12, with respect to Base_12, averaged across all benchmarks.

6.2.3.1 Trends among slipstream-based steering

A key observation is that the percent IPC degradation of all the slipstream-based steering algorithms remain approximately the same as we move from a dual-cluster configuration to a quad-cluster configuration (note that percent IPC degradation is with respect to Base_8 and Base_12, respectively). This can be seen in Figure 6-43, for the *vortex* benchmark. The slipstream-based steering algorithms are on the x-axis and percent IPC degradation is on the y-axis. In the quad-cluster configuration, ineffectual instructions are distributed to a greater extent than in the dual-cluster configuration, for all slipstream-based algorithms. Therefore, ineffectual instructions are delayed more by inter-cluster communication when we move from the dual-cluster to the quad-cluster configuration. However, the IPC degradation is similar for both configurations, confirming the initial proposal that ineffectual instructions are latency-tolerant.

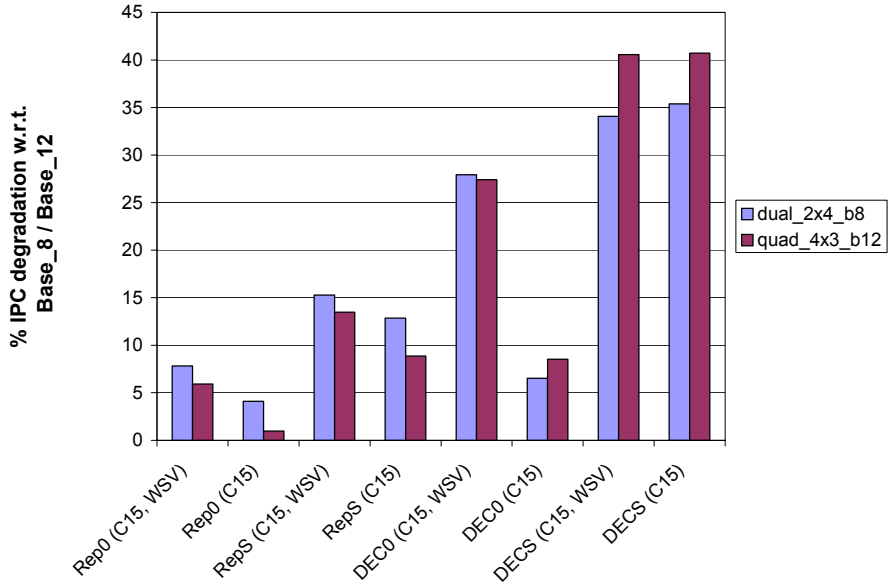

Figure 6-43: Relative performance degradation of slipstream-based algorithms for dual_2x4_b8 and quad_4x3_b12, for *vortex*.

Trends observed for quad_4x2_b8 are repeated for quad_4x3_b12. The slipstream-based steering algorithms that free execution resources by distributing stores perform well compared to the original algorithms. RepS is the best-performing algorithm at high latencies for 6 of the 9 benchmarks (replication is latency-tolerant, while distributing stores relieves issue bandwidth pressure), while DECS has the best performance at low latencies (least issue bandwidth pressure). On average, at a latency of 2 cycles, DECS has a relative performance of 83%, compared to 81% for RepS, 80% for DEC0, and 77% for Rep0 (see Figure 6-53). At 16-cycle latency, RepS has a relative performance of 77% compared to 76% for DEC0, 76% for Rep0, and 61% for DECS. As expected, the relative performance of Rep0 drops the least (only 1%) when latency is increased from 2 to 16 cycles.

6.2.3.2 Comparison of slipstream-based steering with conventional steering

The relative performance of both slipstream-based and dependence-based steering heuristics improve as we move from quad_4x2_b8 to quad_4x3_b12. However, slipstream-based steering heuristics gain more from the increased issue bandwidth per cluster (from 2 per cycle in the previous configuration to 3 per cycle in this configuration). At a 2-cycle latency, DepR has a relative performance of 83% on quad_4x2_b8 (see Figure 6-41) and 86% on quad_5x3_b12 (see Figure 6-53), while RepS has a relative performance of 72% on quad_4x2_b8 and 81% on quad_4x3_b12. Slipstream-based steering increases the number of instructions executed on a cluster. Therefore, the availability of extra execution bandwidth per cluster helps improve performance significantly. Another benefit is that, compared to quad_4x2_b8, the crossover point at which slipstream-based steering outperforms dependence-based steering occurs much earlier.

From Figure 6-54, at 8-cycle latency, Rep0, RepS, and DEC0 improve IPC with respect to dependence-based steering by 6%, 10%, and 10%, respectively. At 16 cycles, Rep0 improves IPC by 31%, RepS by 32%, and DEC0 by 31%, on average.

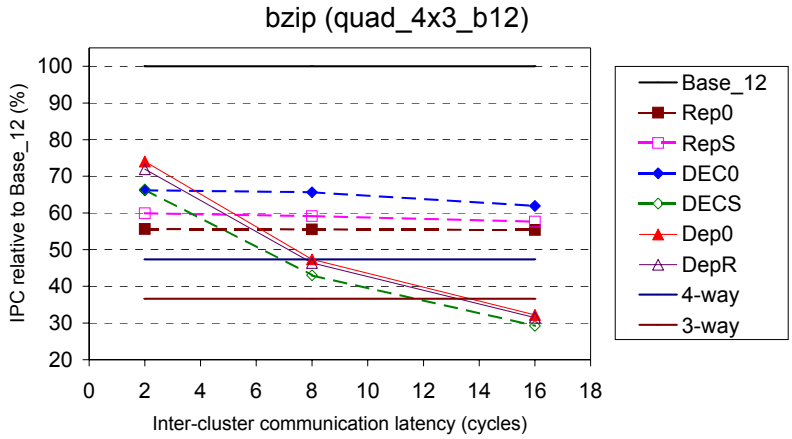

Figure 6-44: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *bzip*.

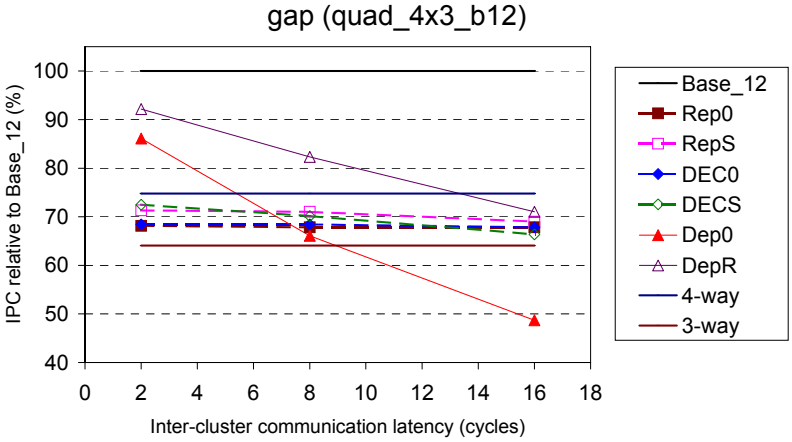

Figure 6-45: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *gap*.

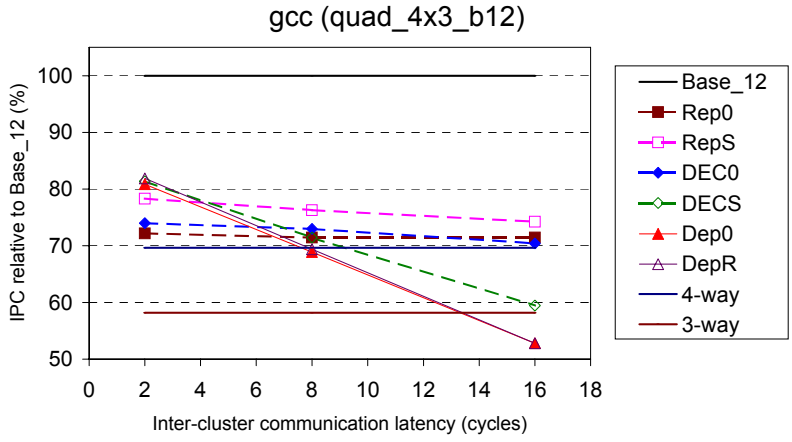

Figure 6-46: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *gcc*.

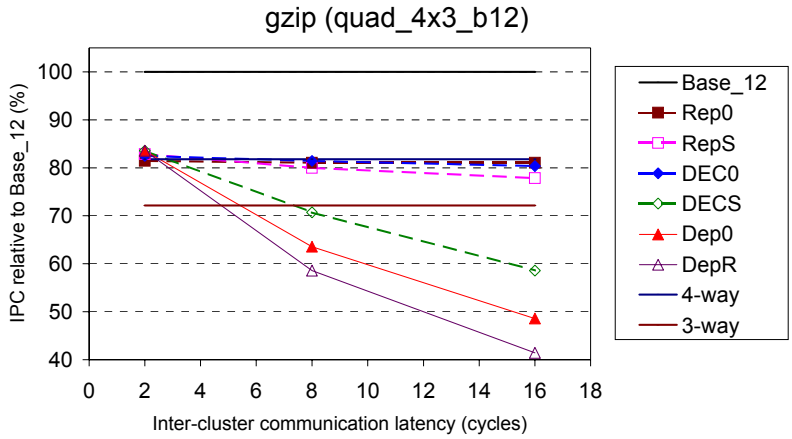

Figure 6-47: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *gzip*.

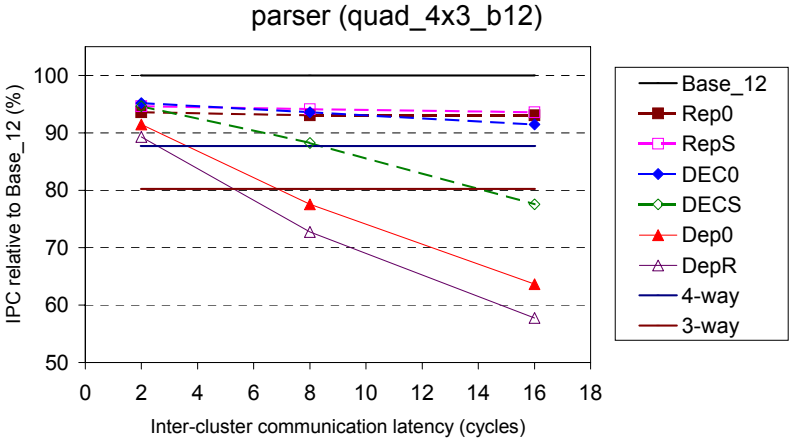

Figure 6-48: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *parser*.

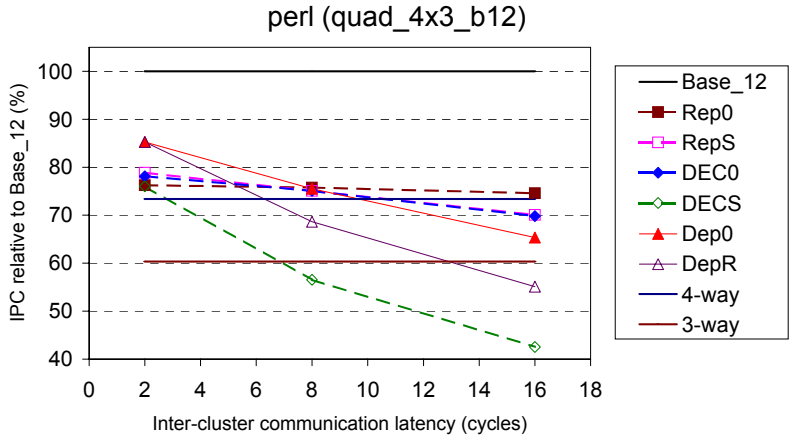

Figure 6-49: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *perl*.

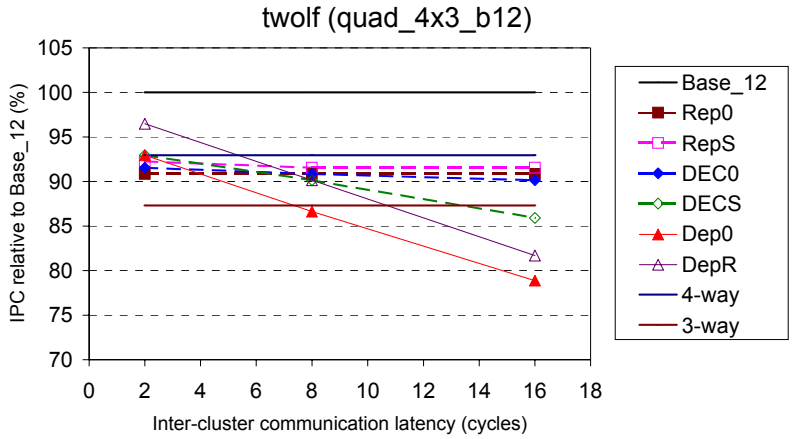

Figure 6-50: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *twolf*.

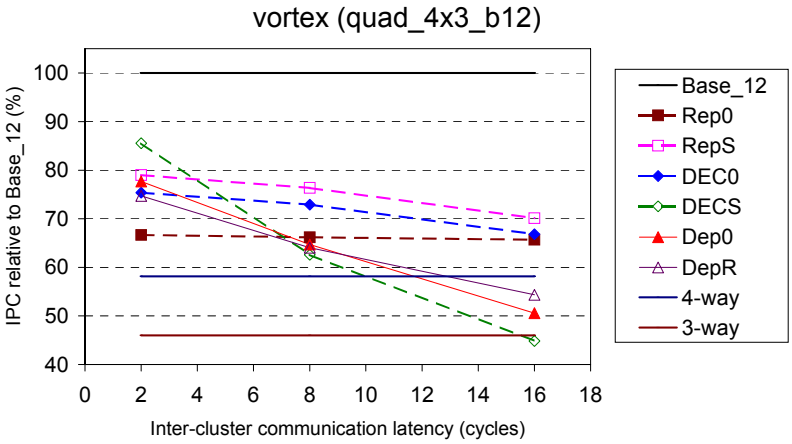

Figure 6-51: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *vortex*.

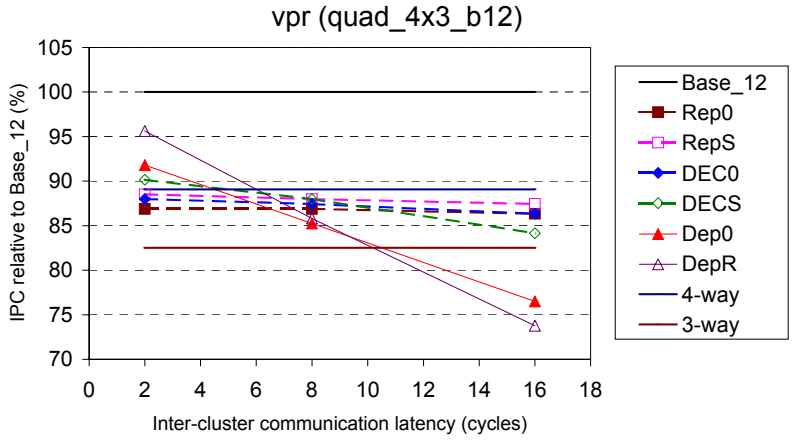

Figure 6-52: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12, for *vpr*.

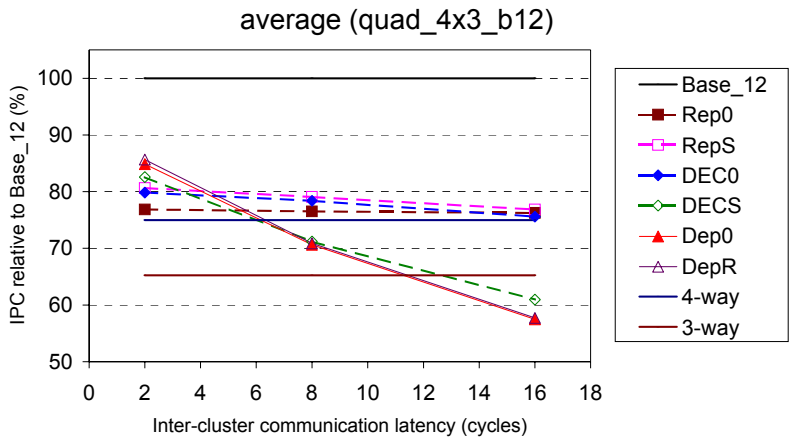

Figure 6-53: Relative performance of slipstream-based steering (confidence threshold of 15 and no WSV) and dependence-based steering on quad_4x3_b12, with respect to Base_12. Results are averaged across all benchmarks.

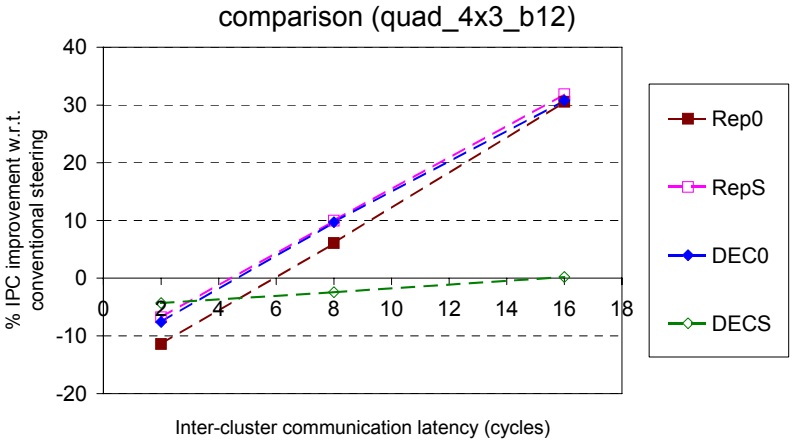

Figure 6-54: IPC improvement of slipstream-based steering (confidence threshold of 15 and no WSV) with respect to dependence-based steering on quad_4x3_b12. Results are averaged across all benchmarks.

7 Related Work

Sastry, Parlacharla, and Smith [27] proposed augmenting the floating-point cluster of a processor for integer execution. They also proposed various compile-time partitioning algorithms for the same.

Parlacharla, Jouppi, and Smith [18] quantified the delay of key processor structures and identified which structures will become more critical with technology scaling. They concluded that it will not be possible to increase processor complexity in the future without adversely affecting cycle time. They proposed a dependence-based architecture that features multiple in-order FIFOs for reducing complexity. They also introduced the concept of clustered microarchitectures with dispatch-driven and execution-driven steering. They studied the performance of several dispatch-driven and execution-driven steering heuristics for a dual-cluster microarchitecture with single-cycle inter-cluster bypass latency.

Farkas, Chow, Jouppi, and Vranesic [9] proposed the multicluster architecture in which each cluster contains an issue queue, a small number of function units, and a subset of the register file. The compiler-managed architectural registers are either local (they map to the register file of a single cluster) or global (they map to the register file of all clusters). In certain cases, the hardware dispatches a copy of an instruction to all clusters (one master copy and multiple slave copies). The master copy does the actual computation, while the slave copy enables the movement of operands and results across the clusters. A static scheduling algorithm is used to assign instructions to the clusters.

Ranganathan and Franklin [22] compare the performance of various decentralized microarchitectures. They study performance as a function of the number of processing elements. They also study the effect of the inter-connection network topology on performance.

Ranganathan and Franklin [23] proposed the PEWS (Parallel Execution Windows) microarchitecture for simplifying logic associated with a monolithic window. PEWs simplifies window logic by splitting the monolithic instruction queue among multiple windows, much like a clustered microarchitecture. They considered a ring interconnection network and a crossbar network for their simulations.

Aggarwal and Franklin [1] studied the scalability of instruction steering algorithms with an increase in the number of clusters. They found that algorithms that work well for four or fewer clusters do not scale to more than four clusters. They model inter-cluster communication latency as a function of the number of clusters. Therefore, they only indirectly study the effect of increasing inter-cluster communication latency on performance. They do not explicitly study the effect of increasing the inter-cluster communication latency for a fixed clustered microarchitecture configuration. Simultaneously increasing the number of clusters and the latency makes it difficult to isolate the effect of individual factors (aggregate issue width, load balance, inter-cluster communication latency, etc.).

Canal, Parcerisa, and Gonzalez [6][7] proposed dynamic steering heuristics for utilizing an idle floating-point cluster for use by integer operations. They proposed adaptive and non-adaptive steering heuristics. Non-adaptive steering uses only immediately available run-time information, whereas adaptive steering also uses history.

Parcerisa and Gonzalez [19] and Rotenberg, Jacobsen, Sazeides, and Smith [25] studied the effect of applying value prediction in a clustered microarchitecture and trace processor, respectively, thereby desensitizing overall performance to the latency for broadcasting global values. By making a value prediction locally within a cluster, inter-cluster dependences are broken, and the delay for communicating global values is exposed only in the case of mispredictions. We propose using the slipstream paradigm for value prediction purposes in a clustered microarchitecture (the WSV criterion).

Clustering is a common technique used in single-register-file VLIW processors. Inter-cluster communication is typically managed via explicit copy instructions. Other models for inter-cluster communication have also been proposed. The design space in clustered VLIW architectures is well researched [17][26]. Many processors in the DSP/embedded domain use a clustered microarchitecture. Good examples are Texas Instrument's TMS320C6000 [29] and Analog's TigerSharc [11].

Baniasadi and Moshovos [4] conducted a detailed study of steering heuristics for quad-cluster superscalar processors. They studied the performance of adaptive and non-adaptive steering heuristics with one-cycle and two-cycle inter-cluster communication

latencies. They also studied the effect of adding two additional stages in the front-end pipeline. They found that a modulo scheme performs best for inter-cluster communication latencies of one or two cycles. However, we observed that dependence-based algorithms perform the best for the dual-cluster as well as the quad-cluster configurations at an inter-cluster communication latency of 2 cycles.

Tune, Liang, Tullsen, and Calder [30] proposed a critical-path predictor and a number of steering heuristics based on instruction criticality. According to one heuristic, all critical-path instructions are steered to a single cluster. Two other dependence-based algorithms were augmented with critical-path information for efficient steering. They compared the performance of the usual steering heuristics with steering heuristics that utilize critical-path information for dual-clustered and quad-clustered microarchitectures. They used a fixed two-cycle latency. The approach in which all critical-path instructions are sent to the same cluster does not perform well compared to the dependence-based approaches (not augmented with critical-path information), whereas our approach in which all effectual instructions are sent to a single cluster does well. We believe one chief reason is that we considered significantly longer inter-cluster latency.

Fields, Rubin, and Bodik [10] also proposed a critical-path predictor that they use to guide instruction steering. They augment a dependence-based algorithm with critical-path information for steering purposes. They also conducted studies similar to the ones conducted by Tune et. al. [30].

8 Summary and Future Work

8.1 Summary

The performance of a clustered microarchitecture suffers primarily because of inter-cluster communication between instructions. Specifically, inter-cluster communication between critical-path instructions is the most harmful. The slipstream paradigm identifies critical-path instructions in the form of effectual instructions. This thesis proposes eliminating virtually all inter-cluster communication among effectual instructions, simply by ensuring that the entire effectual component of the program executes within a cluster. Two execution models are proposed: the *replication model* and the *dedicated-cluster model*. In the replication model, a copy of the effectual component is executed on each of the clusters and the ineffectual instructions are shared among the clusters. In the dedicated-cluster model, the effectual component is executed on a single cluster (the *effectual cluster*), while all ineffectual instructions are steered to the remaining clusters. In the replication model, there is inter-cluster communication solely among ineffectual instructions. In the dedicated-cluster model, there is inter-cluster communication among ineffectual instructions, and ineffectual instructions also wait for values from the effectual cluster.

Based on the replication model, two new steering heuristics have been proposed: *Replication of effectual component (Rep0)* and *Replication of effectual component with store distribution (RepS)*. In Rep0, the effectual component (except effectual branches) is executed redundantly on all clusters, while ineffectual instructions (in addition to

effectual branches) are shared among the clusters. RepS is a variation on Rep0. Store-load dependences are potentially more latency-tolerant than register dependences, therefore effectual stores (in addition to effectual branches and ineffectual instructions) are distributed to reduce pressure exerted on cluster issue bandwidth.

Based on the dedicated-cluster model, two new algorithms have been proposed: *Dedicated cluster for effectual component (DEC0)* and *Dedicated cluster for effectual component with store distribution (DECS)*. In DEC0, the effectual component is executed on a single cluster called the *effectual cluster*, while all ineffectual instructions are steered to the remaining clusters. DECS is a variation on DEC0. In order to ease the pressure on the execution resources of the effectual cluster, effectual stores (in addition to ineffectual instructions) are steered to the remaining clusters.

IPC of the replication model on dual and quad clusters is virtually independent of inter-cluster communication latency. IPC decreases by 1.3% and 0.8%, on average, for a dual-cluster and quad-cluster microarchitecture, respectively, when inter-cluster communication latency increases from 2 cycles to 16 cycles. In contrast, IPC of the best-performing dependence-based steering decreases by 35% and 55%, on average, for a dual-cluster and quad-cluster microarchitecture, respectively, over the same latency range. For dual clusters and quad clusters with low latencies (fewer than 8 cycles), slipstream-based steering underperforms conventional steering because improved latency tolerance is outweighed by higher contention for execution bandwidth within clusters. However, the balance shifts at higher latencies. For a dual-cluster microarchitecture,

dedicated-cluster-based steering (DEC0) outperforms the best conventional steering on average by 10% and 24% at 8 and 16 cycles, respectively. For a quad-cluster microarchitecture, replication-based steering (RepS) outperforms the best conventional steering on average by 10% and 32% at 8 and 16 cycles, respectively.

Slipstream-based steering desensitizes the IPC performance of a clustered microarchitecture to tens of cycles of inter-cluster communication latency. As feature sizes shrink, it will take multiple cycles to propagate signals across the processor chip. For a clustered microarchitecture, this implies that with further scaling of feature size, the inter-cluster communication latency will also increase. Thus, if individual clusters are clocked faster, at the expense of increasing inter-cluster communication latency, performance of a clustered microarchitecture using slipstream-based steering will improve considerably as compared to a clustered microarchitecture using conventional steering.

8.2 Future Work

We have identified several performance/power optimizations for the replication model.

In the replication model, all replicas of an effectual instruction broadcast their identical values to other clusters. We could eliminate communication of these redundant values, and thereby reduce power consumption on inter-cluster bypasses (contention for bypasses also reduced).

On the other hand, inter-cluster communication of redundant values in the replication model can be used for easing pressure on the resources of a cluster. If the value produced by one copy of an instruction becomes globally available before other copies of the same instruction have executed, it would be advantageous to not execute the other copies, as the result is already available in all cluster register files. This will reduce the load on the execution resources of a cluster. We plan to implement a technique whereby copies are squashed from the instruction queues if it is detected that another copy of the same instruction has produced and communicated its value to all other clusters.

The replication models can be exploited for fault tolerance. Since replicated instructions produce identical values, the values produced by different clusters can be compared to confirm that all clusters produce the same value. If any of the values differ, then the program can be rolled back to its non-corrupted architectural state. The entire execution engine of the clustered microarchitecture is provided a level of fault tolerance. There is also coverage of the inter-cluster bypasses, because any transient/permanent faults that occur while values are communicated can be detected (effectual component only).

For the replication model, an IR-predictor/IR-detector pair can be associated with a cluster for predicting the effectualness of instructions on that cluster. Therefore, there can be multiple IR-predictor/IR-detector pairs (equaling the number of clusters), as

opposed to a single IR-predictor/IR-detector pair currently used. This means that an IR-predictor/IR-detector pair is fed instructions from its associated cluster and an instruction can be predicted to be effectual on one cluster and ineffectual on another. A higher percentage of instructions can be predicted ineffectual with this method. Thus fewer instructions will be executed on each cluster. More importantly, we envision a paradigm that eliminates inter-cluster communication altogether, except in the case of IR-mispredictions.

We would like to compare the performance of slipstream-based steering to critical-path-based steering. We would also like to augment the performance of dependence-based steering heuristics with effectual/ineffectual information.

We assumed that the fetch unit is redirected in the cycle after a mispredicted branch is resolved. We also assumed that all clusters squash their instruction queues in the same cycle in which the branch misprediction is resolved. In other words, inter-cluster communication latency is not modeled with respect to branch misprediction recovery. As part of future work, we plan to evaluate recovery latency in the context of a clustered microarchitecture, and propose new microarchitectures accordingly.

We observed that the best-performing slipstream-based steering heuristic depends on the application, the configuration of the clustered microarchitecture, and the inter-cluster communication latency. Currently, the steering heuristic for a clustered microarchitecture is fixed. In order to optimize performance under variable conditions,

we plan to change the steering heuristic at run-time, based on program behavior, inter-cluster communication latency, and configuration of the clustered microarchitecture. This would be beneficial for (1) a clustered microarchitecture that handles programs with varied behavior, (2) a clustered microarchitecture with dynamic voltage scaling (DVS) within clusters, where the inter-cluster communication latency (in cycles) varies with cluster frequency/voltages, (3) a processor in which the number of clusters can be dynamically tuned [3], or (4) a processor with a combination of the above features.

Bibliography

[1] A. Aggarwal and M. Franklin. An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors. *2nd International Symposium on Performance Analysis of Systems and Software*, November 2001.

[2] V.Agarwal, M. S. Hrishikesh, S.W. Keckler, D. Burger. Clock Rate versus IPC: The end of the Road for Conventional Microarchitetcures. *27th International Symposium on Computer Architecture*, June 2000.

[3] R. Balasubramonian, S. Dwarladas, D. H. Albonesi. Dynamically Managing the Communication-Parallelsim Trade-off in Future Clustered Processors. 30th *International Symposium on Computer Architecture*, June 2003.

[4] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. *33rd International Symposium on Microarchitecture*, December 2000.

[5] D. C. Burger, T. M. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[6] R. Canal, J. M. Parcerisa, and A. González. A Cost-Effective Clustered Architecture. *8th International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[7] R. Canal, J. M. Parcerisa, and A. González. Dynamic Cluster Assignment Mechanisms. *6th International Symposium on High Performance Computer Architecture*, January 2000.

[8] G. Z. Chrysos and J. S. Emer. Memory dependence predictions using store sets. *25th International Symposium on Computer Architecture*, June 1998.

[9] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. *30th International Symposium on Microarchitecture*, December 1997.

[10] B. Fields, S. Rubin, and R. Bodik. Focussing Processor Policies via Critical-Path Prediction. *28th International Symposium on Computer Architecture*, June 2001.

[11] J. Fridman and Zvi Greefield. The TigerSharC DSP Architecture. IEEE Micro, pp. 66 – 76, January-February 2000.

[12] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. *29th International Symposium on Microarchitecture*, December 1996.

[13] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. *16th International Conference on Computer Design*, December 1998.

[14] J. J. Koppanalil. A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors. M.S. Thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, May 2002.

[15] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Western Research Laboratory, June 1993.

[16] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. *24th International Symposium on Computer Architecture*, June 1997.

[17] E. Ozer, S. Banerjia, T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. *31st International Symposium on Microarchitecture*, November 1998.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *24th International Symposium on Computer Architecture*, June 1997.

[19] J. M. Parcerisa, A. González. Reducing Wire Delay Penalty through Value Prediction. *33rdInternationalSymposium on Microarchitecture*, December 2000.

[20] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. *33rd International Symposium on Microarchitecture*, December 2000.

[21] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream Memory Hierarchies. Technical Report CESR-TR-02-3, Center for Embedded Systems Research, North Carolina State University, February 2002.

[22] N. Ranganathan and M. Franklin. An Empirical Study of Decentralized Execution Models. 8th *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[23] N. Ranganathan and M. Franklin. The PEWs microarchitecture: reducing complexity through data-dependence based decentralization. Microprocessors and Microsystems 22 Pg 333 – 343, 1998.

[24] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Technical Report, North Carolina State University, November 1999.

[25] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace Processors. *30th International Symposium on Microarchitecture*, December 1997.

[26] J. Sánchez and A. González. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. 33*rd International Symposium on Microarchitecture,* December 2000.

[27] S. S. Sastry, S. Parlacharla, and J. E. Smith. Exploiting Idle Floating-Point Resources for Integer Execution. *1998 ACM Conference on Programming Language Design and Implementation,* June 1998.

[28] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

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

[30] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. *7th International Symposium on High Performance Computer Architecture*, January 2001.

Appendix

Figure A-1 through Figure A-10 show the performance of slipstream-based steering (with and without the WSV criterion for confidence thresholds of 3 and 15) on dual_2x4_b8, with respect to Base_8, for gap, gcc, perl, twolf, and vortex. C3 (WSV) indicates a confidence threshold of 3 and WSV criterion are used, while C3 indicates that the WSV criterion is not used.

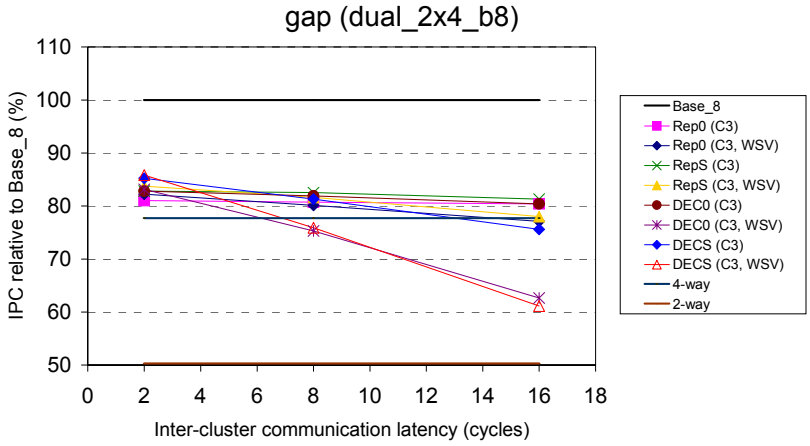

Figure A-1: Relative performance of slipstream-based steering (confidence threshold of 3) on dual_2x4_b8, with respect to Base_8, for *gap*.

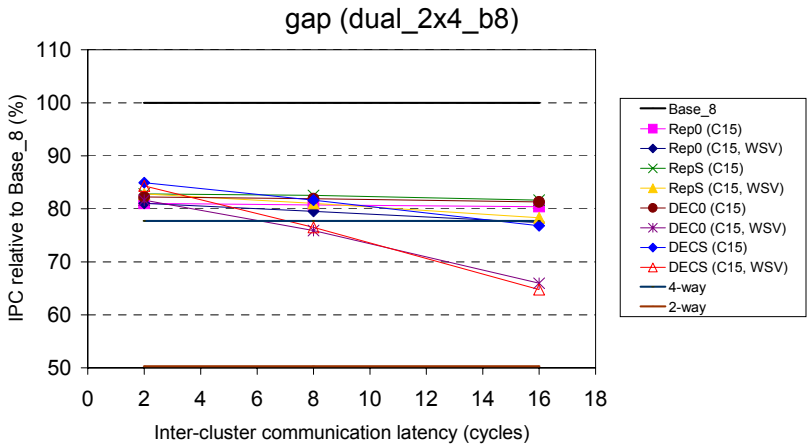

Figure A-2: Relative performance of slipstream-based steering (confidence threshold of 15) on dual_2x4_b8, with respect to Base_8, for *gap*.

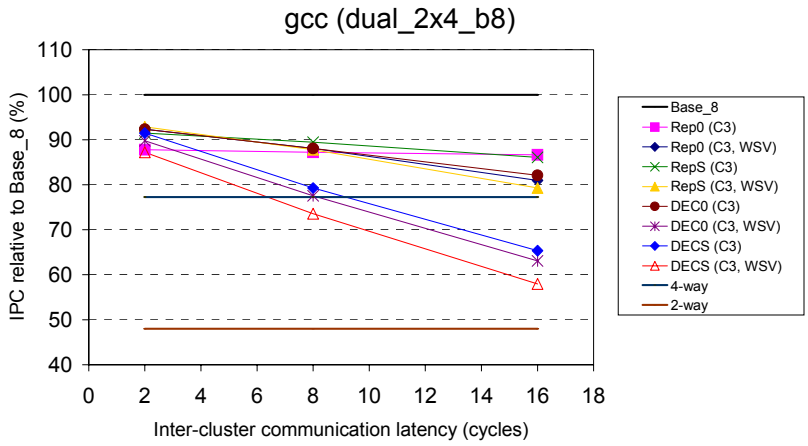

Figure A-3: Relative performance of slipstream-based steering (confidence threshold of 3) on dual_2x4_b8, with respect to Base_8, for *gcc.*

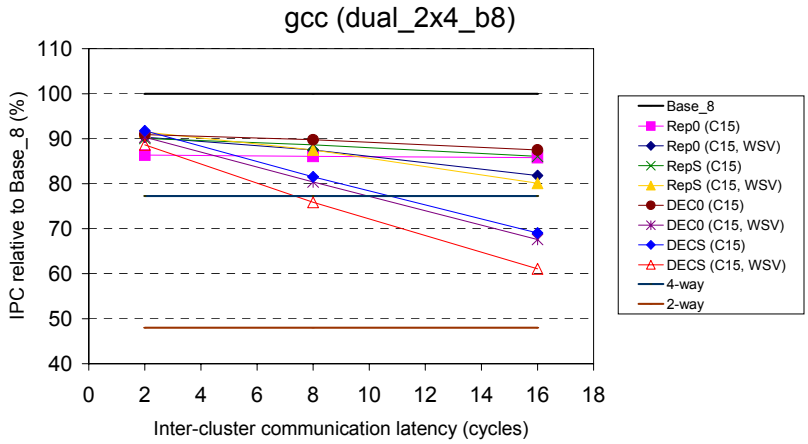

Figure A-4: Relative performance of slipstream-based steering (confidence threshold of 15) on dual_2x4_b8, with respect to Base_8, for *gcc.*

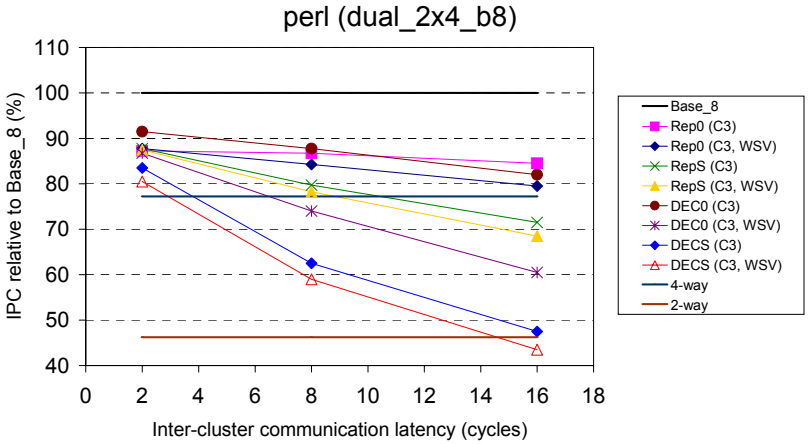

Figure A-5: Relative performance of slipstream-based steering (confidence threshold of 3) on dual_2x4_b8, with respect to Base_8, for *perl.*

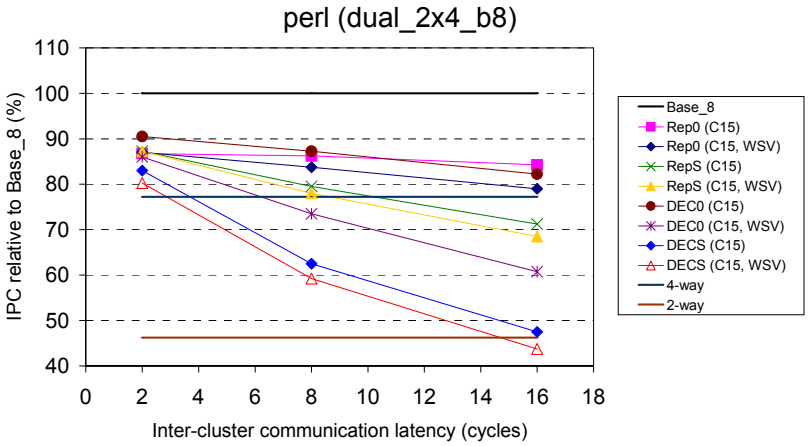

Figure A-6: Relative performance of slipstream-based steering (confidence threshold of 15) on dual_2x4_b8, with respect to Base_8, for *perl*.

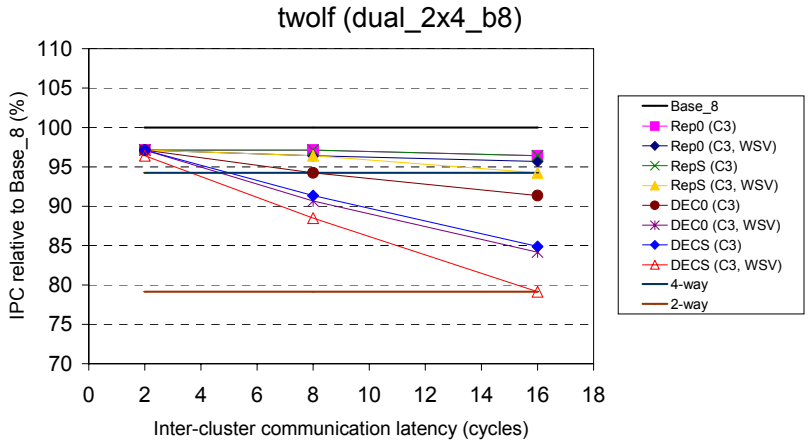

Figure A-7: Relative performance of slipstream-based steering (confidence threshold of 3) on dual_2x4_b8, with respect to Base_8, for *twolf*.

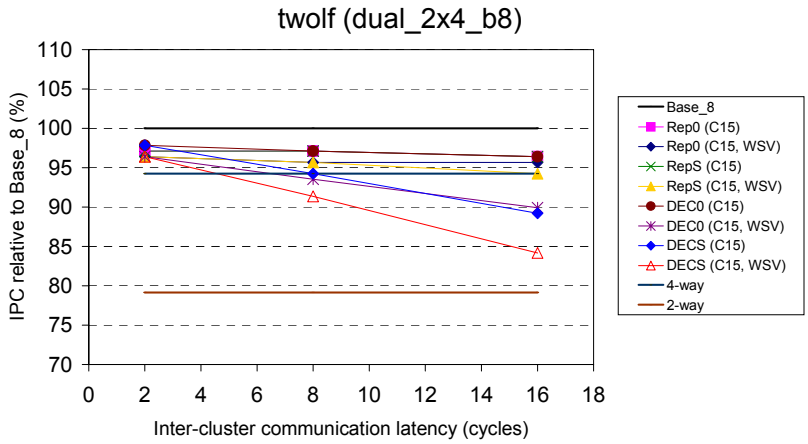

Figure A-8: Relative performance of slipstream-based steering (confidence threshold of 15) on dual_2x4_b8, with respect to Base_8, for *twolf*.

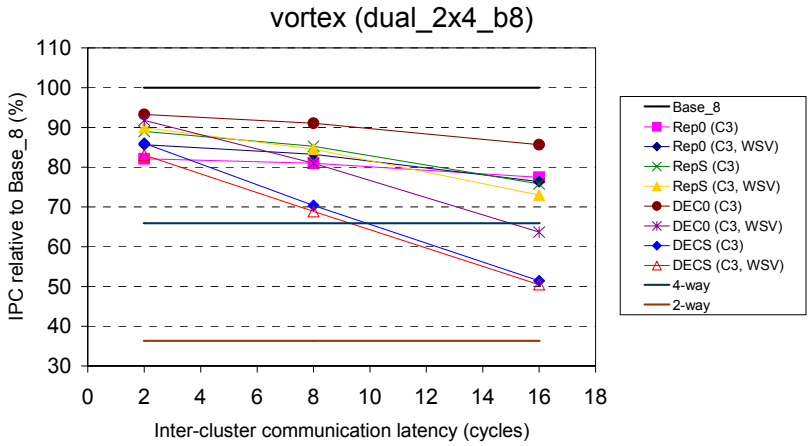

Figure A-9: Relative performance of slipstream-based steering (confidence threshold of 3) on dual_2x4_b8, with respect to Base_8, for *vortex*.

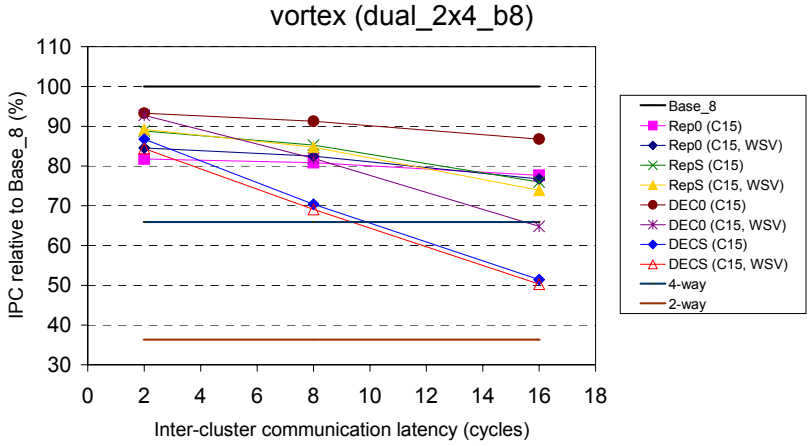

Figure A-10: Relative performance of slipstream-based steering (confidence threshold of 15) on dual_2x4_b8, with respect to Base_8, for *vortex*.

Figure A-11 and Figure A-12 show the performance of slipstream-based steering (with and without the WSV criterion for a confidence threshold of 15) on quad_4x2_b8, with respect to Base_8, for perl and vortex, respectively.

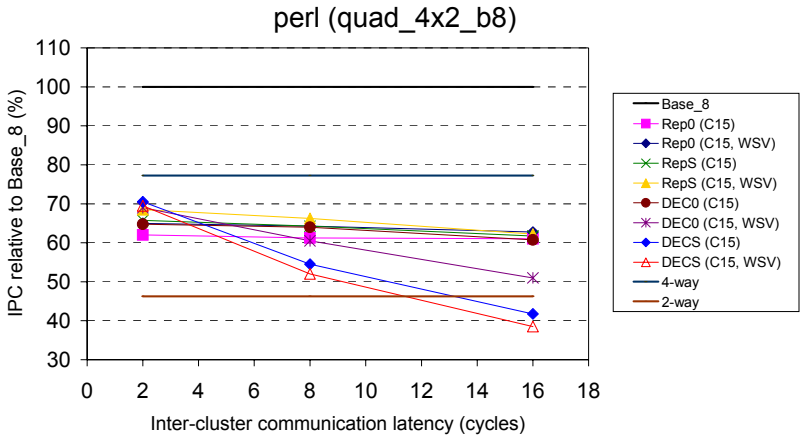

Figure A-11: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x2_b8, with respect to Base_8, for *perl*.

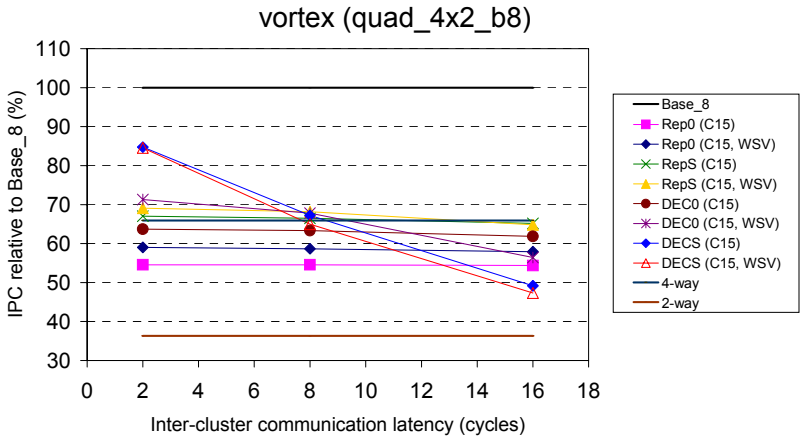

Figure A-12: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x2_b8, with respect to Base_8, for *vortex*.

Figure A-13 and Figure A-14 show the performance of slipstream-based steering (with and without WSV criterion for a confidence threshold of 15) on quad_4x3_b12, with respect to Base_12, for gap and gcc, respectively.

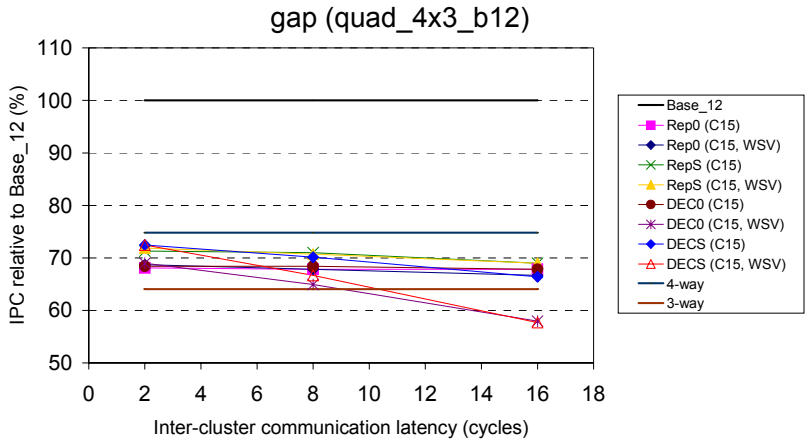

Figure A-13: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x3_b12, with respect to Base_12, for *gap*.

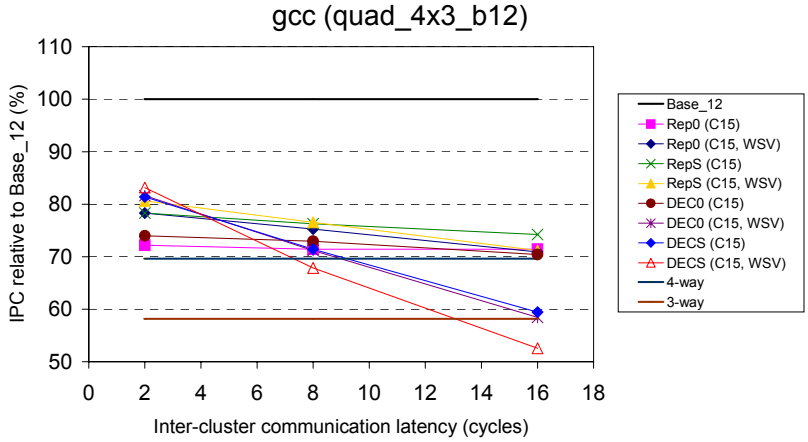

Figure A-14: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x3_b12, with respect to Base_12, for *gcc*.

Figure A-15 through Figure A-18 show the performance of slipstream-based steering (with and without the WSV criterion for confidence thresholds of 3 and 15) on quad_4x3_b12, with respect to Base_12, for perl and vortex.

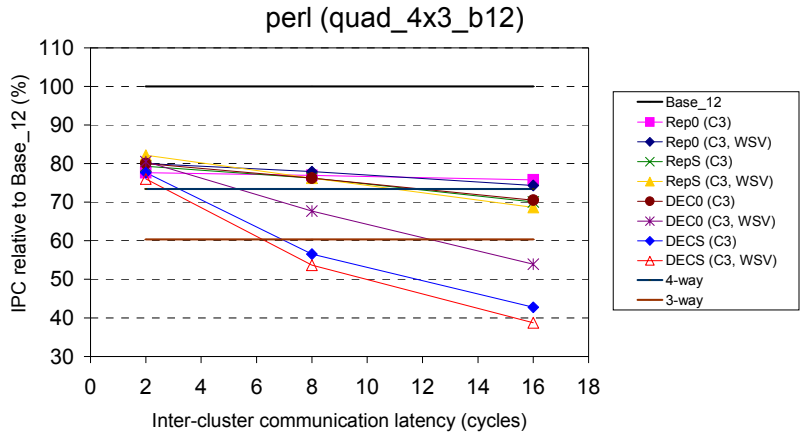

Figure A-15: Relative performance of slipstream-based steering (confidence threshold of 3) on quad_4x3_b12, with respect to Base_12, for *perl.*

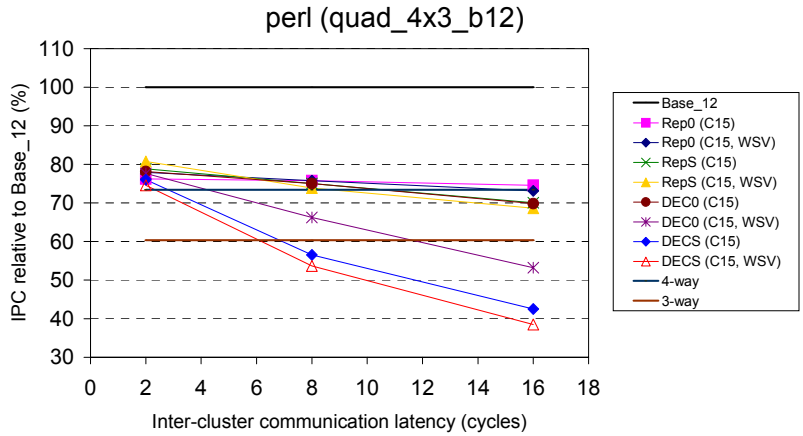

Figure A-16: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x3_b12, with respect to Base_12, for *perl.*

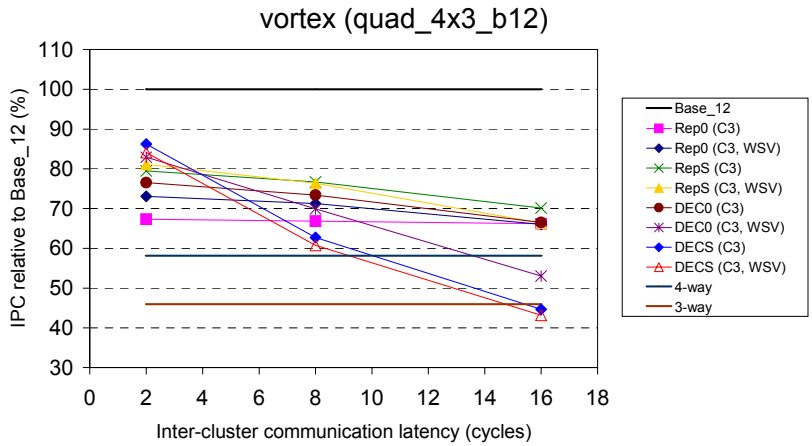

Figure A-17: Relative performance of slipstream-based steering (confidence threshold of 3) on quad_4x3_b12, with respect to Base_12, for *vortex.*

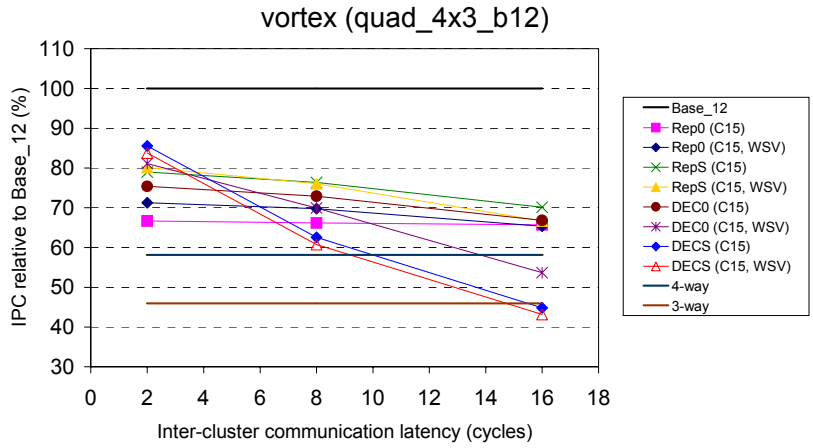

Figure A-18: Relative performance of slipstream-based steering (confidence threshold of 15) on quad_4x3_b12, with respect to Base_12, for *vortex*.