# Balancing Clustering-Induced Stalls to Improve Performance in Clustered Processors

Amirali Baniasadi
ECE Department, University of Victoria
3800 Finnerty Rd.
Victoria, BC, V8P 5C2, Canada
amirali@ece.uvic.ca

## ABSTRACT

Clustered processors lose performance as a result of clustering-induced stalls. Such stalls are the result of distributed resources and cluster communication delays. Our performance analysis of clustered architectures shows how previously proposed methods reduce one group of stalls at the expense of the other. Moreover, we extend previous work and present a new class of cluster assignment heuristics for high-performance clustered processors. We affirm that it is possible to improve performance in clustered processors by taking a more balanced approach towards clustering-induced stalls. Our techniques rely on estimating and predicting resource utilization for clustered processors. We show that, on average, our best technique reduces the performance gap between a dual-clustered and a centralized processor down to 6.9% and 9.2% for 8-way and 6-way processors and for a representative subset of SPEC2K benchmarks.

## Categories and Subject Descriptors

C.1.1 [Single Data Stream Architectures] Pipeline processors.

## General Terms

Design

## Keywords

Clustered Processors, Clustering Stalls

## 1. INTRODUCTION

Exploiting instruction-level parallelism has facilitated rapid performance improvements. One possible way to maintain the steady improvement is to design and develop more complex processors capable of executing more instructions every cycle. Finding more independent instructions may call for searching a larger instruction pool which in turn requires a larger instruction window. However, previous studies show that scaling existing centralized windows may not be possible without adversely affecting clock cycle and consequently performance[1,2,3].


*CF'05*, May 4-6, 2005, Ischia, Italy.


One way to improve ILP while maintaining a fast clock is clustering. In clustering, which is used by designs such as ALPHA 21264[6], multiple smaller instruction windows replace a larger window. As a result, clustered processors give up scheduling flexibility to achieve lower complexity[1] and faster clock. This results in stalling instructions or clustering-induced stalls. Such stalls are either the result of inefficient resource distribution (e.g., issue bandwidth) or of inter-cluster communication latencies.

In contrast to a centralized configuration, in clustered processors, each cluster is limited to only a fraction of the total issue slots per cycle (for example, each of the dual clusters can issue only 4 instructions of the total of 8 per cycle). Accordingly, it is possible for an otherwise ready-to-issue instruction to get stalled in one cluster while free issue slots exist in other clusters. Moreover, since we assume that it takes additional cycles to propagate results across clusters, it is possible for an instruction to get stalled waiting for data that is currently available at another cluster.

In this work we make two contributions:

- First, we perform a detailed analysis of clustered architecture and study where and how performance is lost. We show how previously suggested methods fail to take into account all factors impacting performance.
- Second, we use our findings in the first part to maximize performance through appropriate distribution methods. We show that by taking a more balanced approach it is possible to close the average performance gap between a non-clustered processor and a dual-clustered one by 24% (from 9.1% to 6.9%) for an 8-way and by 22% (from 11.8% to 9.2%) for a 6-way processor respectively.

The rest of the paper is as follows. In section 2 we briefly discuss a number of trade-offs relevant to the design of instruction distribution methods and discuss our model of a dual clustered processor. In section 3 we present our heuristics. In section 4 we report methodology and results. In section 5 we present related work. Finally, in section 6 we conclude and offer closing remarks.

## 2. Distribution Trade-Offs

In this Section, we present a model of a class of dual cluster microarchitectures and discuss the trade-offs involved in developing effective instruction distribution methods for them. Throughout this paper we assume the model of a *uniform* dual cluster organization shown in figure 1. The front-end delivers instructions which are then dispatched to the two clusters via a distribution mechanism. Previous work [1,5,11,12,14] has introduced several distribution mechanisms. Reportedly, this assignment process can dramatically impact performance.

Each cluster is capable of accepting up to $C_{IW}$ instructions per cycle from the distribution mechanism. We assume that once an instruction is assigned to a cluster the decision is final (an alternative would be to decouple execution resources and schedulers as done in the dual cluster 21264 [6])**.** Each cluster has its own set of functional units including data cache ports. Dependent instructions can issue back-to-back provided that they reside in the same cluster. However, propagating results across clusters incurs an additional delay. Finally, all clusters are identical.

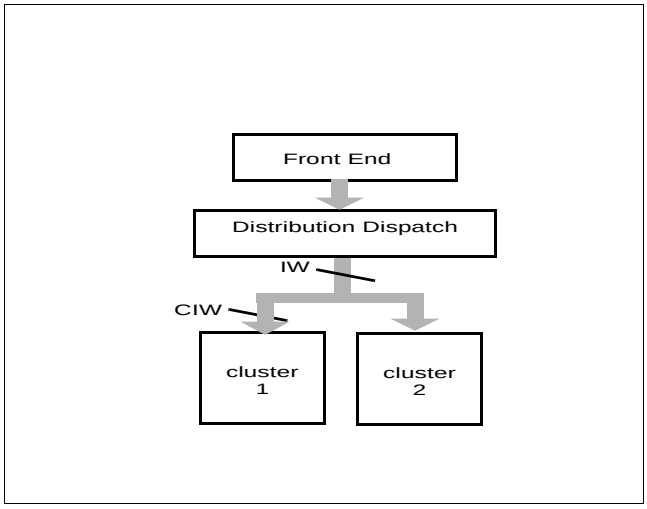


***Figure 1: The dual cluster processor model used in this work.***

In this study, we aim at improving performance by introducing appropriate distribution methods. Note that a perfect and ideal schedule, while desirable, may practically be impossible. Alternatively we aim at improving performance by reducing clustering-induced stalls compared to an equivalent centralized architecture equipped with the same number of resources. Clustering-induced stalls are either *issue-slot* related or *communication latency* related. That is, they are either the result of limited per cluster issue bandwidth (and in general, resource distribution including functional units) or of inter-cluster communication latencies.

As a result of resource distribution in our clustered architectures each cluster is limited to only a fraction of the total issue slots per cycle (for example, each of the two clusters can issue only 4 instructions of the total of 8 per cycle). Accordingly, it is possible for an otherwise ready-to-issue instruction to get stalled in one cluster while free issue slots exist in other clusters. In this case we incur an *issue-slot related* stall. Moreover, since we assume that it takes additional cycles to propagate results across clusters, it is possible for an instruction to get stalled waiting for data that is currently available at another cluster. In this case, we incur a *communication related* stall.

## 3. PERFORMANCE ANALYSIS

In this section, we analyze and provide better insight to clustered microarchitecture's performance. How and where clustered processors lose performance depends on many factors including the scheduling technique exploited. Previous work has introduced several scheduling techniques. We take a closer look at three of the better performing techniques. Later we will use our findings to improve performance.

The first previously proposed method studied here is Firstfit (FF) [14]. Instructions are assigned to the same cluster as long as the cluster has space. Once a cluster is full, instructions are assigned to the next cluster. It is important to note that FF makes no explicit attempt to minimize neither communication- nor issue-induced stalls. Nevertheless, dependent instructions tend to be close in the instruction stream. This often results in reducing the communication-stalls.

The second previously proposed method is Advanced Register Mapping Based Steering (ARMBS)[5]. This method aims at assigning dependent instructions to the same cluster as their parents as long as there is reasonable workload balancing. If an instruction has multiple parents that are assigned to different clusters we pick the cluster holding the least number of instructions. If there is a significant imbalance, instructions are assigned to the least balanced cluster.

The third previously proposed method is the *modulo n* ($MOD_n$)[14]. In this method, instructions are assigned to clusters in a modulo *n* fashion. However while the best modulo value may vary per benchmark, previous study shows that[14], three is the best comprise for this class of methods. In the MOD3 method, and for a dual-cluster processor, the first three instructions are assigned to the first cluster, the second three to the second cluster, the third group again to the first cluster and so on.

To better understand where performance is lost, in figure 1, we report the fraction of committed instructions that are delayed due to clustering for each method in 6-way (part a) and 8-way (part b) dual-cluster processors. Bars from left to right report for FF, ARMBS and MOD3 for a representative subset of SPEC 2000 benchmarks studied here (abbreviations are shown under the "Ab." column in table 1.) We report both the percentage of instructions waiting for a result from a different cluster (lower bar) and those delayed since issue-bandwidth was unavailable in their cluster while available elsewhere (upper bar).

Of particular concern is the relative fractions of instructions delayed due to communication or issued-bandwidth. First, we discuss the 6-way processor. As reported, for the 6-way machine, for FF, most instructions are delayed due to insufficient issue-bandwidth (upper bar). Meantime, not many instructions are delayed due to cluster communication. This is the result of the fact that dependent instructions tend to appear close in the code sequence. However, instructions that appear close in the instruction stream are also more likely to issue about the same time. Consequently, for FF, a large number of instructions are stalled due to per-cluster issue restriction. For ARMBS and MOD3, however, we observe a different kind of behavior, *i.e.*, most instructions are delayed due to cluster communication (lower bar).

In general, in the 8-way processor (reported in part b) stall distribution is similar to that observed in the 6-way processor. Again, FF reduces the communication-stalls effectively but at the expense of high percentage of issue-stalls. Also, ARMBS and MOD3 reduce issue-stalls more effectively but they increase the number of communication-stalls.

Note that ARMBS reduces the issue-stalls as dependent instructions cannot issue at the same cycle. Therefore the issue width is utilized more uniformly. MOD3 also benefits from a

relatively balanced work load as it switches between the clusters regularly.

Also note that, in general, the percentage of issue-stalls is lower for the 8-way processor compared to the 6-way processor. This is caused by the larger number of issue slots available to each cluster in the 8-way machine.

We conclude from figure 1 that state-of-the-art methods have an unbalanced approach. In FF, when assigning instruction streams to the same cluster we benefit from lower communication at the expense of lower resource utilization. In ARMBS and MOD3 we benefit from higher resource utilization at the expense of higher communication.

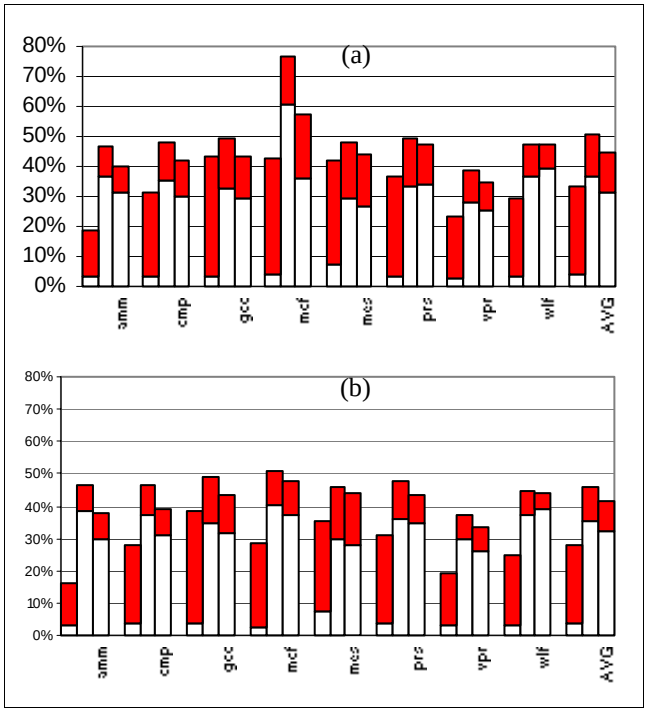


***Figure 2: Percentage of committed instructions that are delayed due to cluster communication (lower bar) and per-cluster issue bandwidth restriction (upper bar) for a) 6-way and b) 8-way dual-cluster processors. Bars from left to right report for FF, ARMBS and MOD3 for the benchmarks studied here. Abbreviations are shown under the Ab. column in table 1.***

While clustered architectures lose performance as the result of the stalls, not always stall frequency indicates performance loss. In fact, it is not strictly true that an effective distribution mechanism should minimize all stalls. To be precise, it is only those stalls that impact the critical computation path that are really important. It may be possible to tolerate some other stalls. Therefore, as we report later, an accurate evaluation of scheduling techniques should rely on performance measurement.

# 4. SCHEDULING HEURISTICS

Our goal is to improve processor performance by taking a more balanced approach and by reducing both kind of clustering-induced stalls simultaneously. Our work relies on estimating and predicting resource utilization in each cluster.

The fact that FF reduces so many communication-stalls encourages assigning consecutive instructions to the same cluster. However, to avoid increasing the number of issue-stalls, we need to assure that not many instructions with the same issue-time are assigned to the same cluster. To do so we aim at assigning as many as consecutive instructions to the same cluster as long as per-cluster issue bandwidth is not overutilized. Key to the success of our techniques is accurate estimation of issue bandwidth utilization.

In this work we introduce two heuristics. In both we aim at assigning maximum number of consecutive instructions to the same cluster as long as we are confident that the issue-bandwidth is not saturated. In the first method we use current issue-bandwidth utilization to do so. In the second method we predict future issue-bandwidth usage by estimating instruction issue time.

## 4.1 Issue Slot Utilization (ISU)

Initially, in this technique we assign consecutive instructions to the first cluster. Meantime, we monitor the number of instructions issuing in the cluster. We continue assigning instructions to the cluster so long the cluster issue-width is not entirely used. Once, and as soon as the issue width is fully utilized, we switch to the next cluster. We repeat the same process in the new cluster, *i.e.*, we keep assigning instructions to the same cluster as long as the issue width is underutilized and switch to the next cluster once the condition is no longer true.

Our goal is to switch to the new cluster only if resources are already fully utilized in the current cluster. However, since there is a gap between the time that instructions are assigned to different clusters, *i.e.*, the dispatch time, and the time they issue, we may witness resource overutilization occasionally for ISU during this time period. To address this issue we introduce the prediction-based technique discussed in the next section.

## 4.2 Issue Slot Utilization Prediction (ISP)

If we had an oracle and knew in advance when an instruction will issue, then a heuristic for cluster assignment would be as follows: as instructions are being dispatched and assigned to each cluster, keep a record of the number of instructions issuing at future cycles in each cluster. Later on, and before assigning instructions to each cluster, use the instruction issue time to check if at the time there will be issue slots available in the cluster. In case all issue slots will be taken at the time in the cluster, check the next cluster.

Of course, we cannot have such an oracle. Instead, we estimate the issue time of instructions as they are dispatched. This is straightforward: we associate a completion time with every register. As instructions are dispatched we predict how long they will take to execute once issued (*i.e.*, we "predict" the latency of the associated functional unit). We also obtain the maximum completion time for its source operands. Adding the two we obtain an estimate for when this instruction will complete. Later for each instruction we check the completion time for its parents. An instruction issue time is estimated to be the maximum of the two parents' completion time. In case both parents have completed at the time of instruction dispatch, we assume that the instruction will immediately issue once dispatch is over.

Once the issue-time is estimated for an instruction we check to find which cluster has available issue slots at the time. If all

clusters will be out of issue slots at the predicted time, we pick the cluster holding the fewest number of instructions. Simply put, we continue assigning instructions to the same cluster as long as the instructions have issue slots available at the time they will issue.

To implement this we store the number of instructions predicted to issue at every cycle in a 1024-entry circular buffer. This allows us to store information for a total of 1024 consecutive cycles. Each entry holds two counters, one per cluster. To decide the buffer size we measured the average time an instruction stays in the pipeline and the average number of occupied reservation stations for all benchmarks. We used the product of the two numbers divided by the number of issue slots to approximate the number of buffer entries needed. We have observed that 1024 entries provide adequate space to store the required information across all benchmarks.

At the event of branch mispredictions we squash this buffer and re-fill it as instructions arrive. This is done to avoid wrong path instructions impacting our future decisions.

# 5. METHODOLOGY AND RESULTS

In this section, we present our analysis of our methods. We report performance results for 6- and 8-way dual-cluster machines. In all cases our base case is a centralized non-clustered machine with the same number of resources. We also report performance for previously suggested methods, *i.e.*, FF, ARMBS and MOD3, and show that our techniques improve performance over them. To provide better insight we also report stall distribution.

We used programs from the SPEC'2k suite compiled for the MIPS-like architecture used by the Simplescalar v3.0 simulation tool set[13]. We used GNU's *gcc* compiler (flags: -O2 –funroll-loops –finline-functions). In the interest of space, we use the abbreviations listed under the "Ab." column in table 1 to refer to these benchmarks. We also report the IPC for the benchmarks studied here for 8-way and 6-way non-clustered processors. The benchmark set studied here incudes different programs including high and low IPC and those limited by memory, branch misprediction, etc. We simulated 1B of the instructions after skipping the initialization. The main architectural parameters of our clustered processor models are shown in table 2.

We simulate high-performance processor models which are motivated by the need to run high IPC codes. Nevertheless, rapid execution of low-IPC code is still necessary in such machines. Therefore, we also include low-IPC programs in the benchmark set.

## 5.1 Performance

Clustering is used in high performance designs in order to achieve higher operating frequency. Nevertheless, we need to know how much faster the clock rate of the clustered architecture has to be (vs. the centralized architecture's clock rate) to result in higher performance. Accordingly, in this section we report performance slowdowns compared to a non-clustered architecture assuming the same clock frequency. Moreover, we assume that the non-clustered architecture has the same overall resources. These slowdowns can serve as bounds on how much faster the clock cycle of the clustered implementation must be.

**Table 1: Benchmarks IPC for 8-way and 6-way non-clustered processors.**

| Benchmark | Ab. | IPC(8-way) | IPC(6-way) |
|---|---|---|---|
| ***ammp*** | amm | 0.63 | 0.62 |
| ***compress*** | cmp | 1.95 | 1.85 |
| ***gcc*** | gcc | 1.60 | 1.51 |
| ***mcf*** | mcf | 1.42 | 1.36 |
| ***mesa*** | mes | 4.02 | 3.57 |
| ***parser*** | prs | 1.73 | 1.63 |
| ***vpr*** | vpr | 2.06 | 1.89 |
| ***wolf*** | wlf | 1.33 | 1.27 |

**Table 2: Base configuration details.**

| ***Base Processor Configuration*** | |
|---|---|
| ***Branch Predictor*** | 32K GShare+32K bi-modal w/ 32K selector |
| ***Scheduler*** | 128 entries, RUU-like |
| ***Fetch Unit*** | 8-way: Up to 8 instr. 6-way: Up to 6 instr. per cycle. Max 2 branches per cycle, 64-entry Fetch Buffer |
| ***Load/Store Queue*** | 8-way:64 entries,4 loads/stores per cycle<br>6-way:64 entries 3 loads/stores per cycle |
| ***Issue, Decode, Commit Bandwidth*** | any 8/6 instructions / cycle per cluster in 8-way/6-way processors. Bandwidth distributed uniformly among clusters. |
| ***Functional l Unit Latencies*** | same as MIPS R10000 |
| ***L1 - Instruction /Data Caches*** | 64K, 4-way SA, 32-byte blocks, 3 cycle hit latency |
| ***Unified L2*** | 256K, 4-way SA, 64-byte blocks, 16-cycle hit latency |
| ***Main Memory*** | Infinite, 100 cycles |

Figure 3 reports performance. The base case configurations are presented in table 2. In 3(a) and 3(b) we compare 6-way and 8-way dual-cluster machines with non-clustered centralized processors equipped with the same execution bandwidth. We also include results for FF, ARMBS and MOD3 for the sake of comparison. Bars from left to right show relative performance for FF, ARMBS, MOD3, ISU and ISP methods. As reported in part (a), for the dual-cluster 6-way machine, on average, performance slowdown is 15.3%, 11.7%, 11%, 9.9% and 9.2% for FF, ARMBS, MOD3, ISU and ISP respectively. For the 8-way dual-cluster machine (part b), average performance slowdown is 9.1%, 10.7%, 10%, 7.5% and 6.9% for FF, ARMBS, MOD3, ISU and ISP respectively.

We conclude from figure 3 that our methods improve performance over previously suggested methods. Moreover, figures 3(a) and 3(b) show that, for the benchmarks studied here, both 6-way and 8-way processors, while having minor differences, follow a similar performance trend. On average, ISP outperforms the rest for both processors.

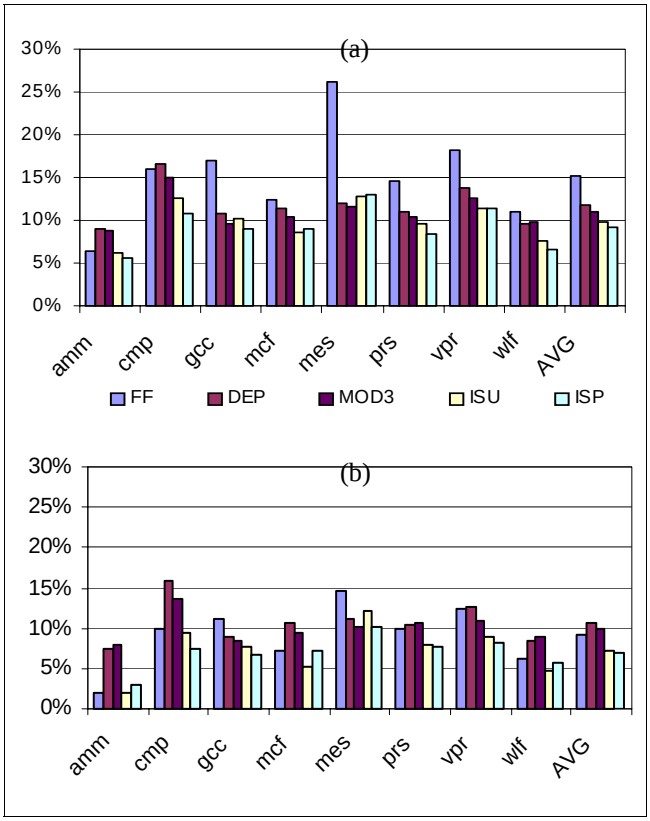


***Figure 3: Performance slowdown for a)6-way b)8-way dual-cluster processors compared to a centralized processor with the same number of resources. Bars from left to right report for FF, ARMBS, MOD3. ISU and ISP. Lower is better.***

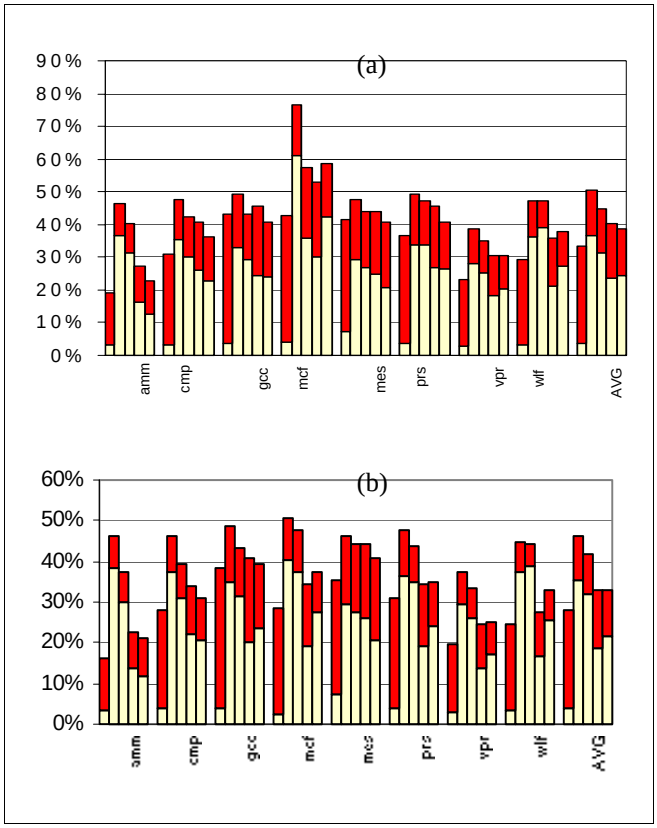


***Figure 4: Percentage of committed instructions that are delayed due to cluster communication (lower bar) and cluster issue bandwidth restriction (upper bar) for a) 6-way and b) 8-way dual-cluster processors. Bars from left to right report for FF, ARMBS, MOD3, ISU and ISP.***

## 5.2 Stall Distribution

As explained earlier one of our goals in this study is to provide a balanced approach which aims at reducing both kind of stalls simultaneously. To evaluate our success, in figures 4(a) and 4(b) we report stall distribution for the dual-cluster 6- and 8-way processors. We include stall distribution for FF, ARMBS and MOD3 for the sake of comparison. Again, we report both the percentage of instructions waiting for a result from a different cluster (lower bar) and those delayed since issue-bandwidth was unavailable in their cluster while available elsewhere (upper bar). Consequently, the entire bar represents the percentage of instructions delayed to clustering. An obvious conclusion from figure 4 is that stall distribution is more balanced for ISU and ISP compared to previously proposed methods. Also, as the entire bar represents, the total number of stalled instructions for ISU and ISP is less than both ARMBS and MOD3 for both the 6- and 8-way processor.We conclude from figure 4 that, as affirmed by figure 3, better performance could be achieved by aiming at minimizing both kind of stalls and by using resource utilization estimation.

## 5.3 Discussion

We conclude from figures 3 and 4 that our methods improve performance over previously suggested methods by effectively reducing both issue- and communication-stalls. However, based on applications response to the techniques discussed here, we realize that not all application perform best under ISP or ISU. In other words, there are applications which either perform best under any of the previously suggested methods or previously suggested methods offer competitive performance.

Applications that perform best under ISU and ISP, benefit from effective reduction in both kind of stalls. In this section we explain why ISP and ISU do not work as effective for some applications.

As reported in figure 3, in the case of the 8-way processor, for 6 of the 8 benchmarks, *i.e., cmp, gcc, prs, vpr, mcf* and *wlf*, either ISP or ISU outperform previous techniques. However, ISP and ISU do not improve performance over all previously suggested methods for *amm* and *mes*. While FF offers competitive performance for *amm*, MOD3 offers competitive performance for *mes*.

In the case of the 6-way processor, the only benchmark that does not perform best under either ISU or ISP is *mes*. For *mes*, MOD3 performs best.

First we discuss why FF does so well for *amm* in the 8-way processor. Later we explain why *mes* performs so well under MOD3 for both 8- and 6-way processors.

*Amm* has the lowest IPC as reported in table 1. Consequently, it has the lowest resource utilization among all benchmarks [15]. Accordingly, there is not much room for optimizations aiming at efficient usage of distributed issue-slots. ISU and ISP aim at avoiding occasions where resources are

underutilized in one cluster while needed in another cluster. For applications with low resource utilization such occasions are less likely to happen as we rarely face resource shortage in any of the clusters. This is specially true for the 8-way processor which contains higher number of resources compared to the 6-way processor. Therefore, in the 8-way processor and for low-IPC applications, a simple method such as FF may perform well since instructions rarely experience issue slot saturation. As the 6-way processor has less number of resources, FF no longer performs as well as it does for the 8-way processor.

As a result of low resource utilization, *amm* has the lowest performance slowdown among all benchmarks. Consistently, *amm* shows the least sensitivity to issue-bandwidth (for *amm* IPC changes from 0.63 to 0.62 when we change the issue-width from 8 to 6 as reported in table 1).

*Mes*, on the other hand, has the highest IPC and sensitivity to issue-bandwidth among all benchmarks (see table 1.). As a result, resource utilization is the highest for this benchmark. Therefore, quite often, issue-slots are fully utilized in both clusters. Consequently, ISU and ISP are forced to switch clusters frequently. Accordingly, for *mes*, we have observed that average number of consecutive instructions assigned to the same cluster for our methods could be as low as two instructions. As a result, the number of communication stalls is increased. Therefore MOD3 (which switches between the two clusters less often as average number of consecutive instructions assigned to the same cluster is 3) performs better. In other words, for *mes*, scheduling techniques can rarely address the resource shortage when there is no free resource available in any of the clusters. Under such circumstances, MOD3 outperforms the rest since it minimizes the number of communication stalls. Recall that as the number of consecutive instructions assigned to a cluster increases, communication stalls start to decrease.

## 5.4 Latency Considerations

The timing overhead associated with ISU should not be an issue since the information needed to decide about switching clusters is available and maintained easily by monitoring the cluster issue width usage. A possible implementation comprises a per cluster global-AND of the utilized flags of the cluster's issue slots and a global current-cluster pointer.

As for ISP, latency impact could be potentially more critical. Note that the circular buffer used by ISP is cycle-indexed. As such, to access this filter, instruction issue time estimation is needed. the register completion time information required by this method can be made available via the register renaming mechanism. The estimated issue time can be computed in parallel with register renaming. Therefore, through this work we assume that implementing ISP would not result in additional latency.

# 6. RELATED WORK

Numerous studies have investigated partitioning as a way of scaling over existing, centralized dynamically-scheduled superscalar architectures. A class of methods aims at extracting parallelism by making large prediction-based steps in the dynamic instruction stream, *e.g.*, [4, 7, 8, 9, 10]. Here we restrict our attention to works that investigated partitioning a traditional architecture.

Palacharla, Jouppi and Smith studied the delay characteristics of key processor structures [1]. They demonstrated that it will not be possible to naively scale existing designs without adverse effects on clock cycle. They proposed using clustering as a solution and studied various non-traditional scheduling mechanisms for dual-clustered architectures (*e.g.*, FIFO-based schedulers). They also suggested the dependence-based method. The ARMBS method studied here should be viewed as a variation of their technique.

Farkas, Chow, Jouppi and Vransevic proposed and studied a dual-clustered architecture along with a cluster-aware static scheduling technique [11]. Canal, Parcerisa and González studied a variety of instruction distribution methods also for a non-uniform dual-clustered architecture [5].

Fields *et al.* [12] used a criticality predictor to improve performance in clustered architectures. They used criticality-based information and modified the ARMBS method to introduce "focused instruction steering". Accordingly, when an instruction has two non-complete parents, they assign in to the cluster of the critical predecessor. We simulated their techniques and compared it to our methods. We do not report results for this methods since, as both our findings and their experiments show, performance-wise, their method performs very similar to the ARMBS method. This is due to the fact that "focused instruction steering" uses the critical path only to break ties, which occur in ARMBS infrequently.

Alpha 21264 uses a dual-cluster integer execution unit[6]. In this processor, the execution integer unit includes four integer pipelines and two integer register files. The unit is divided to two clusters each containing two integer pipes and a single integer register file. The difference between our microarchitecture and that used by Alpha 21264 is that in the latter issue queues are not tied to a specific cluster. In other words, instruction execution clusters (*i.e.*, functional units, register files and cache ports) and schedulers are decoupled. Accordingly, an instruction is first assigned to a scheduler and then, based on input operand availability, is sent to the appropriate execution cluster. However, Alpha 21264 cannot schedule each instruction to all four integer execution units. Rather is statically assigns instructions to two of the fours pipelines before they enter the queue. Compared with the model used here, the Alpha approach provides higher scheduler flexibility at the cost of more complexity. With a centralized scheduler, clustering hardly reduces the complexities associated with wakeup/select logic.

Baniasadi and Moshovos [14] investigated instruction distribution methods for quad-cluster, dynamically-scheduled superscalar processors. They studied a variety of methods including FF and Modulo method also presented here. We have compared our work to their techniques and affirmed that by exploiting resource utilization performance could be further improved.

Balasubramonian *et al.* [16] introduced techniques to match the hardware to application's needs dynamically in clustered architectures. They showed that by gathering program metrics at periodic intervals it is possible to improve performance for reconfigurable clustered architectures. In this work we focus on statically defined architectures.

Aggarwal and Franklin studied the effect of various hardware parameters on the scalability of different instruction distributions including MOD3 and FF [17].

## 7. CONCLUSION

We extended previous work on cluster assignment techniques by introducing a new class of heuristics. In addition, we investigated how our proposed techniques impact performance and the distribution and frequency of clustering-induced stalls. We improved previous methods by estimating application resource utilization and by reducing both kind of stalls simultaneously. On average, our best method (which predicts future resource utilization) reduced the performance gap between the dual-cluster and non-clustered 8-way and 6-way processors to 6.9% and 9.2% for a subset of SPEC 2K benchmarks.

## 8. REFERENCES


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

[2] M. T. Bohr. Interconnect scaling - the real limiter to high performance ULSI. *International Electron Devices Meeting Technical Digest*, 1995.

[3] D. Matzke. Will Physical Scalability Sabotaze Performance Gains?. In *IEEE Computer, 30(9)*, Sept. 1997.

[4] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Annual International Symposium on Microarchitecture-31*, Nov. 1998.

[5] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proc. High Performance Architecture 6*, Jan. 2000.

[6] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 architecture. In *Proc. of International Conference on Computer Design*, Dec. 1998.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. Symposium on Architectural Support for Languages and Operating Systems VIII*, Oct. 1998.

[8] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proc. on Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[9] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. International Symposium on Computer Architecture-22*, June 1995.

[10] J. G. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. High Performance Computer Architecture-4*, Jan. 1998.

[11] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[12] B. Fields, S. Rubin, R. Bodik, Focusing Processor Policies via Critical-Path Prediction. In *Proc. of ISCA-28*, July 2001.

[13] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Technical Report Computer Sciences Tech. Report #1342*, University of Wisconsin-Madison, June 1997

[14] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduling Superscalar Processors. *In Proc. of MCRO-33.* Dec. 2000

[15] . Buyuktosunoglu, A., Schuster, S., Brooks, D., Bose, P., Cook, P. and Albonesi, D., An Adaptive Issue Queue for Reduced Power at High Performance, *Workshop on Power-Aware Computer Systems*, held in conjunction with ASPLOS, November 2000.

[16] R. Balasubramonian, S. Dwarkadas and D.H. Albonesi. "Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors". *In Proc. of ISCA-30.* June 2003.

[17] A. Aggarwal, M. Franklin. "An Empirical Study of Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors". *In Proceedings of ISPASS, 2001*