Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors

Aneesh Aggarwal, Manoj Franklin

***Abstract*— In the evolving sub-micron technology, importance of wire delays is growing, making it particularly attractive to use decentralized designs. A common form of decentralization adopted in processors is to partition the execution core into multiple *clusters*. Each *cluster* has a small instruction window, and a set of functional units. A number of algorithms have been proposed for distributing instructions among the *clusters*. The first part of this paper analyzes (qualitatively as well as quantitatively) the effect of various hardware parameters such as the type of cluster interconnect, the fetch size, the cluster issue width, the cluster window size, and the number of clusters on the performance of different instruction distribution algorithms. The study shows that the relative performance of the algorithms is very sensitive to these hardware parameters, and that the algorithms that perform relatively better with 4 or fewer clusters are generally not the best ones for a larger number of clusters. This is important, given that with an imminent increase in the transistor budget, more clusters are expected to be integrated on a single chip.**

The second part of the paper investigates alternate interconnects that provide scalable performance as the number of clusters is increased. In particular, it investigates two hierarchical interconnects — a single ring of crossbars and multiple rings of crossbars — as well as instruction distribution algorithms to take advantage of these interconnects. Our study shows that these new interconnects with the appropriate distribution techniques achieve an IPC (instructions per cycle) that is 15-20% better than the most scalable existing configuration, and is within 2% of that achieved by a hypothetical ideal processor having a 1-cycle latency crossbar interconnect. These results confirm the utility and applicability of hierarchical interconnects and hierarchical distribution algorithms in clustered processors.

***Index Terms*— Clustered Processor Architecture, Pipeline processors, Interconnection architectures, Load balancing and task assignment**

I. Introduction

Two hardware trends play a key role in processor design: the increasing number of transistors in a chip [5], [22], and the increasing clock speeds. A major implication of going for sub-micron technology is that *wire delays* become more important than *gate delays* [22], especially for long global wires. Another important implication of the physical limits of wire scaling is that centralized structures such as the dynamic scheduler do not scale well [13]. A natural way to deal with the wire delay problem is to build the processor as a collection of independent *clusters*, so that (i) there are only a few *global wires*, and (ii) very little communication occurs through global wires. Fast localized communication can be done using short wires. The design of clustered processors is also much simpler than that of a monolithic processor, because once a single cluster core has been designed, it can be easily replicated for multiple cores as the transistor budget increases.

Aneesh Aggarwal is with the ECE Department at Binghamton University, NY. Manoj Franklin is with the ECE Department at University of Maryland, College Park, MD.

In the recent past, several decentralization proposals and evaluations, using a small number of clusters, have appeared in the literature [2] [3] [4] [6] [7] [8] [12] [13] [14] [15] [16] [18] [20] [19] [21]. Detailed comparative studies of these decentralization approaches for 2-cluster and 4-cluster processors [4] [6] have been pivotal in understanding their behavior with a small number of clusters. With continued increases in the number of transistors integrated on a chip, we can expect more clusters on a single chip [2]. Adding more clusters could potentially improve the performance of a program, and studying the impact of increased number of clusters on the performance of a single thread of execution is very important. Balasubramonian et. al. [2] investigate dynamic reconfiguration for optimal use of a subset of clusters from a large number of clusters. However, as clusters are "turned-on" and "turned-off", the inter-cluster interconnection may also have to be reconfigured for optimal performance. The main focus of their work was to study dynamic reconfiguration for optimal ILP exploitation, whereas in this paper, our main focus is to study the performance of a non-reconfigurable clustered processor with a large number of clusters. To study the performance impact of increased number of clusters in an effective manner, it is important to first study the impact of different hardware parameters such as the type of interconnect, cluster issue and window size, the number of clusters, etc. on the performance of clustered processors. In particular, the following specific questions need to be addressed:

- How well does each instruction distribution algorithm (used to distribute instructions among the clusters) permit scaling of performance?
- What is the impact of the interconnect on performance scaling?
- What are the implications of different design decisions such as processor fetch size, cluster issue width, and cluster window size?

The first part of this paper reports the results of experiments conducted to provide specific, quantitative evaluations of different trade-offs. These studies show that the relative performance of the distribution algorithms is very sensitive to hardware parameters such as fetch size, issue width, and cluster window size. They also show that the distribution algorithms performing relatively well with 4 or fewer clusters generally do not perform well with a larger number of clusters,

and sometimes even perform worse. For scalable performance from a large number of clusters, two important inter-related issues need to be addressed: (i) better algorithms for instruction distribution among the clusters, and (ii) better interconnects for inter-cluster communication.

The second part of this paper investigates alternate inter-cluster interconnects as well as instruction distribution algorithms suitable for a larger number of clusters. In particular, it presents two hierarchical interconnects: a single ring of crossbars and multiple rings of crossbars. Although many interconnects have been studied in the context of parallel processors [9], interconnects for on-chip clustering have not been studied in detail. The important issues in designing the two types of interconnects (on-chip and off-chip) are very different (the primary objectives of off-chip interconnects are fault-tolerance, consistency of data, etc., whereas that of on-chip interconnects is latency). The hierarchical interconnects presented in this paper divide the clusters into groups (internally connected using crossbars), which are connected together using either a single ring interconnect or a multiple rings interconnect. The paper also presents instruction distribution algorithms that take advantage of these hierarchical interconnects. We expect such interconnects to be used in future clustered processors having a large number of on-chip clusters.

The rest of this paper is organized as follows. Section 2 presents a common framework for analyzing the performance of different instruction distribution algorithms. Section 3 presents detailed simulation results and analysis of the different instruction distribution algorithms. Section 4 investigates hierarchical interconnection topologies and distribution algorithms. Section 5 presents an experimental evaluation of these new schemes. Section 6 discusses other advantages that are obtained when using hierarchical interconnects. Finally, Section 7 concludes the paper.

II. Important Issues in Clustering

In this paper, a decentralized processor has several clusters, as shown in Figure 1. Each cluster has a dynamic scheduler (DS) and several functional units (FUs). Instructions from multiple clusters are issued independently of each other, subject only to the availability of operand values. An interconnection network (ICN) connects the clusters together for supporting inter-cluster communication. Instructions are distributed among the clusters using a distribution algorithm and are committed from the clusters in program order.

A. Important Criteria for Performance Scaling

Two important criteria for obtaining a good instruction distribution are: (i) minimize inter-cluster communication, and (ii) maximize load balancing among the clusters. The former attempts to reduce the wait for operands, and the latter attempts to reduce wait for an issue slot.

Inter-Cluster Communication: In a single-cluster processor, when a value is produced, it is immediately available to all the instructions. However, in multiple clusters, some instructions have to wait due to inter-cluster communication. Typically, inter-cluster communication latency increases with the number of clusters, because of the physical distance between the clusters. The relative importance of inter-cluster communication thus increases for more clusters.

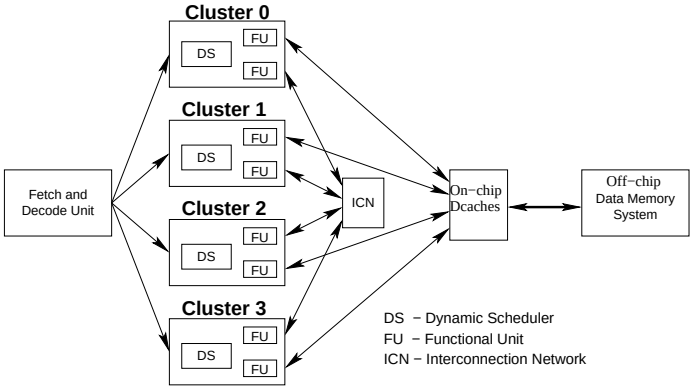

Fig. 1. A Generic 4-Cluster Processor

Load Balancing: With an unbalanced instruction distribution, only modest performance will be obtained, because each cluster can execute only a small number of instructions every cycle. To get good performance, instructions should be distributed somewhat evenly across all clusters. A simple-minded way to handle the load balancing problem is to have a round robin distribution of instructions. Such a scheme was found to be good for a 4-cluster processor [4].

A good distribution algorithm attempts to meet both criteria by placing data-independent instructions in different clusters, and data-dependent instructions in the same cluster.

B. Hardware Factors Affecting Performance Scaling

There are several hardware attributes that affect the performance of a multi-cluster processor. Many of these factors are inter-dependent, and the relative importance of a particular attribute is somewhat related to the specific values used for the remaining factors.

1) Number of Clusters: When a small number of clusters is used, load balancing is more important than inter-cluster communication. This is because the worst-case inter-cluster communication latency is low, and the overall issue width of the processor is small, which has to be utilized properly. With an increase in the number of clusters, the inter-cluster communication latency increases, increasing the importance of inter-cluster communication. Hence, the algorithm that works very well for small number of clusters may not be the best for large number of clusters.

2) Cluster Layout and Inter-Cluster Interconnect: A major factor affecting inter-cluster communication latency is the physical layout of the clusters (because wire delays dominate the communication latency) and the type of interconnect used to connect the clusters. Clearly, the worst-case latency increases with the number of clusters, because of the increase in the physical distance between the clusters.

There are various interconnects that can be thought of in the context of clustered processors, such as bus, crossbar, ring (uni-directional or bi-directional), tree, and grid. The bus is a simple, fully connected network that permits only

one data transfer at any time, and may be a poor choice for carrying out non-trivial inter-cluster communication. A crossbar interconnect also provides full connectivity from every cluster to every other cluster. When using a crossbar, all clusters become logically equidistant to each other. This makes the instruction partitioning algorithm less complex. However, in a crossbar interconnect, all inter-cluster communication becomes "global" in nature, and cross-chip wire delays begin to dominate inter-cluster communication latency.

With a ring interconnect, the clusters are connected as a circular loop, and there is a notion of neighboring and distant clusters. A ring is ideal if most of the inter-cluster communication can be localized to neighboring clusters. Hence, the ring forces distribution algorithms to expend more efforts in localizing inter-cluster communication to neighboring clusters.

3) Processor Fetch Size: In general, performance increases monotonically with the processor fetch size because of the increase in the effective window size. For a clustered processor, increasing the fetch size also reduces the relative importance of load balancing. This is because, when the fetch size is large, more clusters are likely to be filled sooner with instructions. A large fetch size is therefore likely to favor those distribution algorithms geared to reducing inter-cluster communication. However, with very large fetch widths, the performance could saturate because of branch mispredictions.

4) Cluster Issue Width: In general, performance increases monotonically with the cluster issue width although at some point the performance would saturate. As the cluster issue width is increased, the importance of load balancing decreases. This is because, in a particular cycle, even if a cluster has several ready instructions, they are more likely to get executed sooner because of the additional number of issue slots available in each cluster. Thus, when the cluster issue width is large, distribution algorithms geared to reduce inter-cluster communication perform better.

5) Cluster Window Size: If the cluster window size is small, we get load balancing for free, favoring distribution algorithms that focus on reducing inter-cluster communication. As the cluster window size is increased, the relative importance of load balancing increases, and the distribution algorithm has to consciously do load balancing; otherwise, too many instructions may end up in the same cluster.

C. Instruction Stream Distribution Algorithms

Perhaps the most important issue in a clustered processor is the instruction distribution algorithm. An improper distribution could distribute instructions unevenly, or cause too much inter-cluster communication, thereby degrading performance. We only give a brief description of the several hardware-based distribution algorithms that have been proposed. For some of these algorithms, the original versions were described in the references only for two clusters; our description extrapolates this to more clusters. Notice that the compiler can attempt to schedule the instructions in such a way as to maximize the effectiveness of the distribution algorithm used [11]; the investigation of such compiler techniques is beyond the scope of this paper.

First-Fit: In this method [4], instructions are assigned to the same cluster until the cluster fills up. Then, instructions are assigned to the next cluster, and so on. The attraction of this scheme is its simplicity and reduced inter-cluster communication. As dependent instructions are typically close to each other, they end up either in the same cluster or neighboring clusters.

$\mathbf{Mod_n}$: In this method [4], n consecutive instructions are assigned to one cluster, the next n instructions are assigned to the next cluster, and so on. This scheme is suited to situations where load balancing plays a major role such as configurations with smaller fetch sizes, smaller cluster issue widths, and larger cluster window sizes. This algorithm is likely to perform poorly when there is a lot of data dependence between instructions close to each other.

Dependence-based: This scheme [7] partitions the instructions based on the state of the instruction's operands. If the operands are available, the instruction is assigned to the least loaded cluster; if not, it is assigned to one of its operand-producing clusters. However, the hardware does not know if the results of independent instructions serve as operands for a subsequent instruction. If that is the case and these instructions are placed in distant clusters, then performance will be affected. Furthermore, this kind of partitioning is somewhat complex because the partitioning hardware has to consider which clusters have the latest values for each register operand.

Dependence-based with Load Balancing: In this method [7], *dependence-based* partitioning is performed as in the previous scheme. But if a considerable amount of load imbalance is observed among the various clusters, then load balancing is given priority and the instructions are assigned to the least loaded cluster.

Load-Store Slice (LdSt Slice): In this method, all instructions on which a load or a store is dependent on are sent to the same cluster. This scheme also gives some importance to load balancing by assigning the remaining instructions according to the load situation of the clusters. However, in this scheme a value produced by a load might be used by an instruction assigned to a far away cluster. A static partitioning *ldst slice* scheme was proposed in [20], and a dynamic version was proposed in [6].

Strands: In this method [16], off-line hardware first constructs data-dependent strands within a trace, and stores this information in a trace cache. When a trace is later fetched from the trace cache, each pre-identified strand within the trace is allocated to a cluster based on the availability of the operands feeding the strand. This algorithm takes intra-trace data dependences into consideration, and is likely to perform better when using larger traces.

III. Experimental Analysis of Existing Algorithms

A. Experimental Methodology and Setup

Our experimental setup consists of an execution-driven simulator based on the MIPS-II ISA. The simulator does cycle-

by-cycle simulation, including execution along mispredicted paths. The default hardware parameters are given in Table 1. In addition, an 128 KB, 8-way set-associative, 1 cycle access time trace cache [17] is used to store recently seen traces. Memory disambiguation is done using a centralized disambiguator. Our pipeline consists of the following stages: fetch, decode/rename, dispatch, issue, execute, memory, writeback, and commit. In all our experiments, we assume a broadcast-based inter-cluster communication, where each value goes to all the clusters.

The experiments vary 6 parameters: the instruction distribution algorithm, the number of clusters, the cluster interconnect, the fetch size, the cluster issue width, and the cluster window size. While varying a parameter, the rest of the parameters are kept at their default values. No attempt is made to keep the overall hardware resource requirement a constant[1], as a goal of this paper is to investigate how the behavior of each distribution algorithm is dependent on each parameter. For comparison, we also simulate a hypothetical superscalar processor, with an overall scheduler size same as that of the overall scheduler size of the clustered processor it is compared against. That is, the hypothetical superscalar processor used to compare against a 4-cluster processor (with cluster issue width 2 and cluster window size 16) will have a $4 \times 16 = 64$-entry window and an issue width of $4 \times 2 = 8$ instructions/cycle.

For benchmarks, we use a collection of integer programs from the SPEC2000 suite compiled for a MIPS R3000-Ultrix platform with a MIPS C (Version 3.0) compiler using the optimization flags distributed in the SPEC benchmark makefiles. The benchmarks are simulated for 500 million instructions after skipping the first 500 million instructions. Our simulation experiments measure the execution time in terms of the number of cycles required to execute the fixed number of committed instructions (excluding NOPs). While reporting the results, the execution time is expressed in terms of instructions per cycle (IPC).

B. Results with Crossbar/Bi-directional Ring Interconnects

Our first set of experiments are with a crossbar interconnect, keeping the other parameters (except the number of clusters) at their default values. Figure 2 (consisting of 8 graphs, one for each benchmark) plots the results of these experiments. The format is the same for each graph: the X-axis denotes the number of clusters and the Y-axis denotes the IPC; each graph has 7 lines, one for the hypothetical superscalar processor, and the remaining ones for the 6 distribution algorithms (the legend is in the graph for `gzip`). We shall discuss these results along with the results for the bi-directional ring interconnect.

The second set of experiments are with a bi-directional interconnect for the same set of hardware parameters. These results are presented in Figure 3. The format of the graphs is the same as that of Figure 2. The main observations from Figures 2 and 3 are:

- The relative performance of the algorithms vary considerably, based on the interconnect and the number of clusters.
- Algorithms that perform the best for 4 or fewer clusters are not necessarily the best ones for larger numbers of clusters. For example, for 4 clusters, the *ldst slice* algorithm is among the toppers in performance; but for 12 clusters, the *first-fit* is the best performer for 4 of the benchmarks when a ring interconnect is used.
- For 4 clusters, the crossbar interconnect is generally better than a bi-directional ring, because of the 1-cycle communication latency for a crossbar. But as the number of clusters is increased, the performance with crossbar drops below that with the bi-directional ring interconnect. This behavior is the same, irrespective of the distribution algorithms used, and can be attributed to the increasing latency for all inter-cluster communications when using a crossbar. On the other hand, with a bi-directional ring, the communication latency between the neighboring clusters is still 1 cycle, and the distribution algorithms can use this by assigning the data-dependent instructions to the neighboring clusters.
- The performance of most of the algorithms does not scale well (as compared to the hypothetical superscalar) when the number of clusters is increased. With 12 clusters, the IPC obtained with the different distribution algorithms is, on the average, about 40% lower than the hypothetical superscalar's IPC. To bridge this gap, we need better algorithms and/or interconnects, such as the ones proposed in Section 4.

C. Inter-Cluster Communication and Load Imbalance

Figure 4 presents statistics on the average number of instructions stalled in a cycle due to inter-cluster communication and due to load imbalance, when using the crossbar interconnect. An instruction is considered to be stalled due to inter-cluster communication if its operand(s) have already been produced, but at least one operand is in transit from another cluster. A data-ready instruction is considered to be stalled due to load imbalance if there was no free issue slot in its cluster in that cycle, but there was a free slot in a different cluster.

Figure 5 presents similar statistics for the ring interconnect. The X-axis depicts 3 different number of clusters—4, 8, and 12. The Y-axis depicts the average number of instructions stalled per cycle. For each configuration and benchmark, statistics are presented for 3 distribution algorithms—*first-fit*, *ldst slice*, and mod_3. We selected 3 algorithms for clarity of presentation; these three are representative of the 6 algorithms in their behavior.

The first thing we wish to point out from Figure 5 is that *first-fit* almost always has the fewest number of instructions stalled due to inter-cluster communication, but almost always has the largest number of instructions stalled due to load imbalance. When the number of clusters is increased, *first-fit*'s relative benefit in reduced inter-cluster communication increases, whereas its relative loss due to load imbalance remains the same. This attests to why *first-fit*'s IPC continues to improve (for a bi-directional ring), as the number of clusters is increased.

On the other hand, the behavior of mod_3 is diametrically opposite to that of *first-fit*. It almost always has the maximum

[1]Varying multiple parameters at a time (to keep the hardware requirements a constant) will make it very difficult to isolate the contribution of the individual parameters.

Default Values for Processor Parameters		Default Values for Cluster Parameters	
Parameter	Value	*Parameter*	Value
Fetch/Commit Size	16 instructions	*Cluster window size*	16 instructions
Control flow predictor	2-level trace predictor 1024 entry, pattern size 6	*Functional Units*	2 Int., 1-cycle lat. 1 Mul/Div, 10-cycles lat.
L1 - Instruction cache	128K, 4-way set assoc., 2 cycle access latency	*Cluster issue width*	2 instructions/cycle
		Register File	32 entries, 1-cycle acc.
L1 - Data cache	128KB, 4-way set assoc., 2 cycle access latency	*Inter-cluster delay for ring*	1 cycle
L2 - Unified cache	Infinite, 10 cycle access latency	*Inter-cluster delay for crossbar*	MAX($\lceil \log_2 \frac{\#clusters}{2} \rceil$, 1) cycles

TABLE I

DEFAULT PARAMETERS FOR THE EXPERIMENTAL EVALUATION

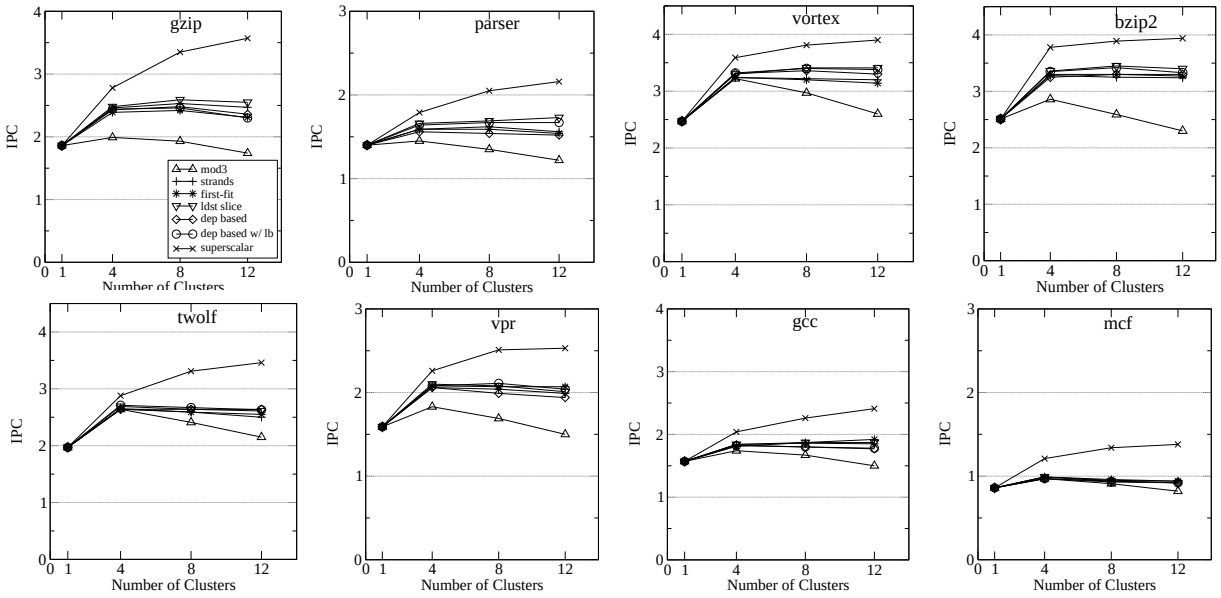

Fig. 2. IPC for Varying Number of Clusters; Fetch size = 16, Cluster issue width = 2, Cluster window size = 16, **Interconnect = Crossbar**

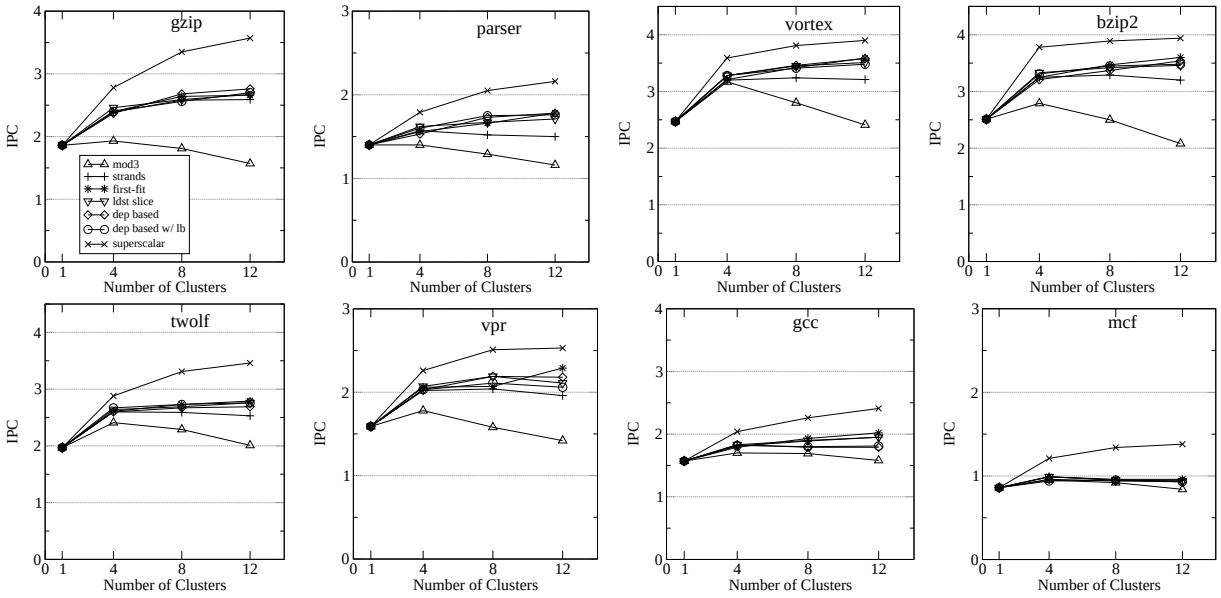

Fig. 3. IPC for Varying Number of Clusters; Fetch size = 16, Cluster issue width = 2, Cluster window size = 16, **Interconnect = Bi-directional Ring**

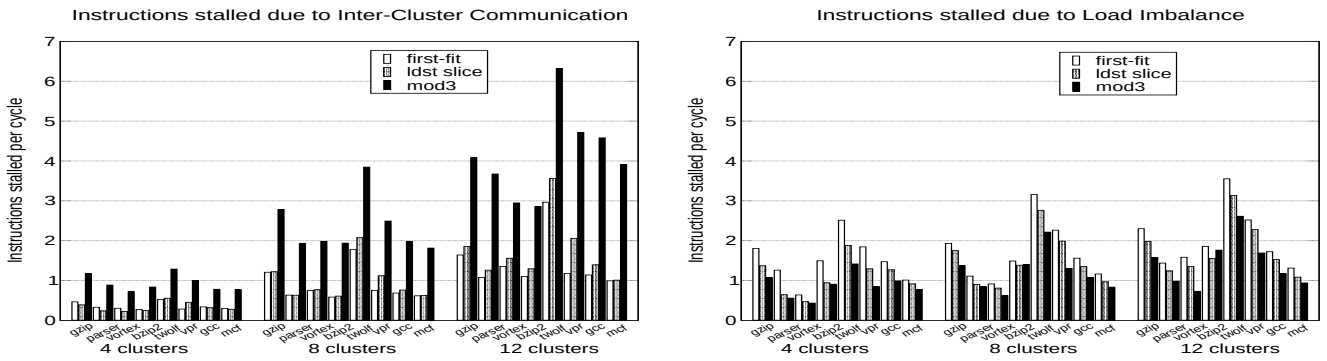

Fig. 4. Average Number of Instructions Stalled per Cycle; Fetch size = 16, Cluster issue width = 2, Cluster window size = 16, **Interconnect = Crossbar**

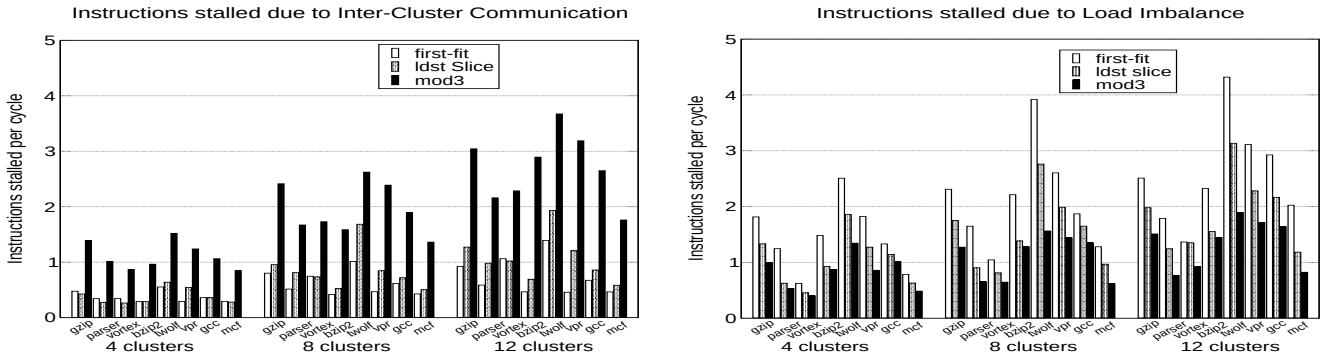

Fig. 5. Average Number of Instructions Stalled per Cycle; Fetch size = 16, Cluster issue width = 2, Cluster window size = 16, **Interconnect = Bi-directional Ring**

number of instructions stalled due to inter-cluster communication and minimum number of instructions stalled due to load imbalance. When the number of clusters is increased, its instruction stall count due to inter-cluster communication increases at an alarming rate, attesting to its poor IPC values. The gains for mod_3 due to load balancing are obscured, because the number of issue slots available over the entire processor increases.

The instruction stall values for *ldst slice* algorithm are almost always in between that of *first-fit* and mod_3. Hence, for a bi-directional ring interconnect with 4 clusters, where both load balancing and inter-cluster communication are important, *ldst slice* has the best performance. For *first-fit*, whatever is gained by less communication (which is not much because of the lower communication delays) is more than lost due to load imbalance.

For a crossbar interconnect, on the other hand, *ldst slice* performs better than *first-fit* and mod_3, because all communications incur the same delays. Hence, as the number of clusters is increased, *ldst slice*, which assigns the data-dependent instructions to the same cluster, performs the best among the three. Although *first-fit* takes care of the intra-trace communication, it suffers from inter-trace communications. The mod_3 algorithm suffers from both inter-trace and intra-trace communication delays.

D. Impact of Cluster Issue Width

To analyze the impact of cluster issue width on the distribution algorithms, experiments are conducted by keeping the number of clusters fixed at 8, and the other parameters at their default values. In the interest of space, we restrict these experiments to a bi-directional ring interconnect. We use 3 different values of issue width—1, 4, and 8. For these sensitivity studies, we simulate only the 3 representative algorithms—*first-fit*, *ldst slice*, and mod_3. Figure 6 presents the results of these experiments. In this graph, the X-axis denotes the cluster issue width, and the Y-axis denotes the IPC. As expected, increases in cluster issue width produce a monotonic improvement in performance for all 3 algorithms. However, the relative effects on the 3 algorithms are different. When the issue width is 1, *first-fit* performs the worst for all benchmarks and mod_3 performs better than *first-fit*. This is because, load balancing plays a major role when issue width is small. When the cluster issue width is increased, the relative importance of load balancing decreases progressively, and the performance of *first-fit* and *ldst slice* improve considerably relative to that of mod_3. *First-fit* outperforms *ldst slice*, because it is more capable of reducing inter-cluster communication.

E. Impact of Cluster Window Size

The experiments are again conducted for 8 clusters, with the other parameters fixed at their default values. We use 3 values of cluster window size—16, 32, and 64. Figure 7 gives the results, where the X-axis denotes the cluster window size and the Y-axis denotes the IPC.

From the graph, we can see that increases in the cluster window size consistently improve the relative performance of mod_3, i.e., the performance gaps between mod_3 and the

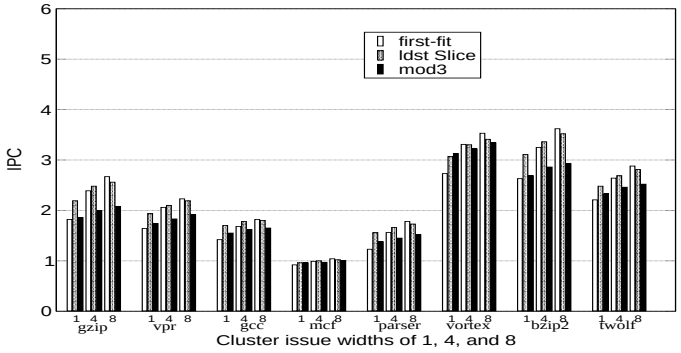

Fig. 6. IPC for **Varying Cluster Issue Width**; Number of Clusters = 8, Cluster window size = 16, Fetch size = 16, Interconnect = Bi-directional Ring

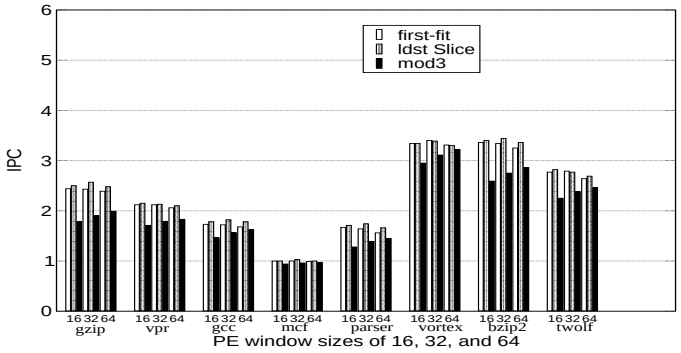

Fig. 7. IPC for **Varying Cluster Window Sizes**; Number of clusters = 8, Cluster issue width = 2, Fetch size = 16, Interconnect = Bi-directional Ring

other algorithms narrow considerably, because as the cluster window size is increased, load balancing becomes relatively more important. The effect of inter-cluster communication still dominates the effect of load balancing, and so mod_3's performance is still below that of the other two algorithms. The performance of *first-fit* and *ldst slice*, on the other hand, generally decreases with increases in the window size, because of their weak ability to ensure load balancing. The performance of *ldst slice* varies in a non-standard fashion when the cluster window size is increased. Its performance improves uniformly for all benchmarks when the window size is increased from 16 to 32, but thereafter it falls. The increase in performance while going from a window size of 16 to 32 is because *ldst slice* attempts to target both inter-cluster communication and load balancing.

F. Impact of Processor Fetch Size

All the parameters (apart from the fetch size) are again kept at the default for a 8 cluster processor. The fetch sizes studied are 8, 12 and 16. Figure 8 presents the results, where the X-axis denotes the fetch size and the Y-axis denotes the IPC. As discussed in Section 2.2, as the fetch size increases, load balancing becomes relatively less of a concern, and performance is mostly determined by inter-cluster communication. Hence the relative performance of *first-fit* and *ldst slice* algorithms (with respect to mod_3) uniformly increases with increases in the fetch size. Another effect that can be seen from the histogram is the drop in the performance of mod_3 in some

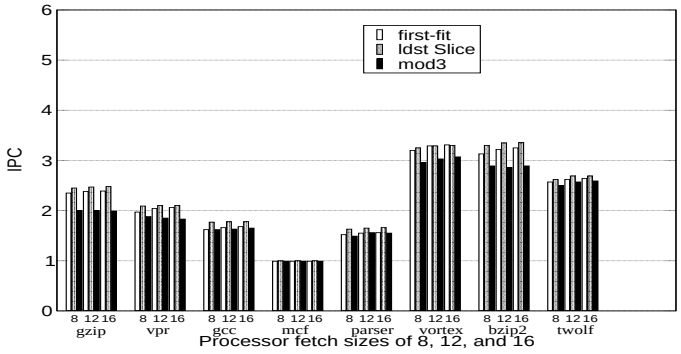

Fig. 8. IPC for **Varying Fetch Sizes**; Number of clusters = 8, Cluster issue width = 2, Cluster window size = 16, Interconnect = Bi-directional Ring

of the cases, as the fetch size is increased. This effect can be attributed to the unpredictable nature of the distribution done by mod_3. Its performance is very sensitive to where exactly the fetch unit cuts the traces. If the fetch unit cuts the trace at one particular point, then mod_3 may end up assigning several data-dependent instructions to the same cluster, and when the trace is cut at some other point, then mod_3 may end up assigning those points to different clusters.

IV. Hierarchical Interconnects and Distribution Algorithms

One of the major reasons for the lack of scalability exhibited by most of the existing distribution algorithms is the increase in inter-cluster communication delays with increase in the number of clusters. Communication delays cannot be decreased arbitrarily. Crossbar interconnects have a limitation that as the number of clusters is increased, the hardware involved makes it very difficult to reduce the inter-cluster communication delays beyond a certain point. For ring interconnects, efforts can be made to limit the communication to within neighboring clusters and incur less overall communication latency. Nevertheless, the communication latency between distant clusters is still very high, hurting the scalability of ring interconnects. To take advantage of both types of interconnects (ring and crossbar), we investigate hierarchical interconnects for on-chip clustering.

A. Hierarchical Interconnects

The basic idea behind hierarchical interconnects is that a small number of physically close clusters are interconnected using a low-latency crossbar and the distant clusters are connected using a ring. Hence, in hierarchical interconnections, the clusters are divided into groups of clusters, internally interconnected by a crossbar. Multiple cluster groups are interconnected by either a bi-directional ring or multiple bi-directional rings. When using such an interconnect along with an appropriate instruction distribution algorithm, most of the inter-cluster communication happens only within the low-latency crossbars. High communication latency may be incurred occasionally when communicating between distant clusters.

1) Single Ring of Crossbars: Figure 9 shows the layout of a ring of crossbars interconnect for 8 clusters and 12 clusters. In this layout, 4 clusters form a group. Each 4-cluster group is internally connected by a 1-cycle latency crossbar, and the groups are connected by a bi-directional ring. The communication latency for inter-group communication is 2 cycles for neighboring groups and an additional cycle for each hop. For example, in Figure 9, for 12 clusters, the communication of a value from a cluster in group 1 to another cluster in group 1 requires 1 cycle, but communication from a cluster in group 1 to a cluster in group 2 or 3 requires 2 cycles.

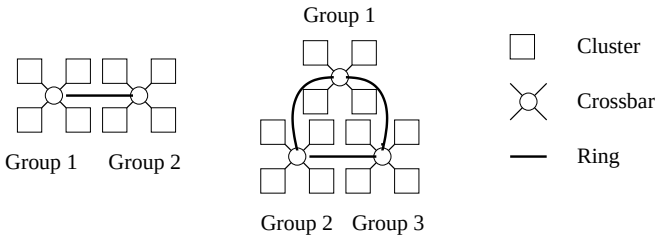

Fig. 9. Single Ring of Crossbars Interconnect for 8 and 12 Clusters

2) Multiple Rings of Crossbars: To reduce the number of values that have to go through multiple hops before reaching their consumers, we investigate multiple rings of crossbars interconnect also (shown in Figure 10). In this layout also, each 4-cluster group is internally connected by a 1-cycle latency crossbar, but the groups are inter-connected using multiple bi-directional rings. Each of the 4 clusters within the 4-cluster group is connected to a corresponding cluster in the neighboring groups. For example, the bottom left cluster of group 2 is connected to the bottom left clusters of groups 1 and 3. In this interconnect, the communication latency across the 4-cluster groups depends on the clusters participating in the communication. If the communication is between the clusters connected together by the ring interconnect, then the latency is 1 cycle; otherwise, a cycle gets added to latency for an additional hop across the 4-cluster group. For example, the communication between the bottom left cluster of groups 1 and 2 requires 1 cycle. But, the communication between the bottom left cluster of group 1 and the bottom right cluster of group 2 requires 2 cycles.

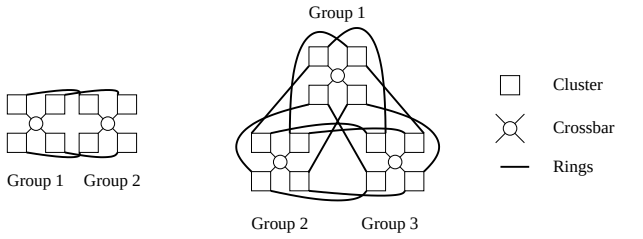

Fig. 10. Multiple Rings of Crossbars Interconnect for 8 and 12 Clusters

B. Instruction Distribution

This subsection discusses instruction distribution algorithms for the hierarchical interconnects.

Single Ring of Crossbars: It was seen that with 4 clusters, the best performance is generally obtained with the *ldst slice* algorithm, with the crossbar interconnect. On the other hand, the performance of the *first-fit* algorithm, with the ring interconnect, monotonically increases as the number of clusters is increased. Using this information, we investigated the following instruction distribution approach.

While distributing the instructions among the clusters, consider only a group of the clusters at any time and optimize the distribution within this group. Once the instruction window in this cluster group fills up, consider the next group of clusters, and so on. For instruction distribution within a cluster group, we use the *ldst slice* algorithm. The top-level distribution algorithm, i.e., the one used to distribute traces across cluster groups, is *first-fit*. The 2-level distribution approach thus attempts to get the benefits of two different algorithms, in the situation where each performs the best.

Rings of Crossbars: For the rings of crossbars interconnect, we use the *ldst slice* algorithm. We also performed experiments using the *2-level* distribution algorithm discussed in the previous paragraph along with other distribution algorithms. We found that the *ldst slice* algorithm performed the best. The reason for this is the presence of the additional connectivity, which restricts most of the communication to 1 cycle latency. The *2-level* algorithm divides the instruction stream into chunks of instructions and tries to avoid communication within a chunk, while paying some communication costs across the chunks. If the communication within a chunk cannot be avoided, then it incurs a communication cost of 1 cycle. *Ldst slice* on the other hand, tries to avoid any form of communication. If it is not able to achieve that, most of the times, it pays a communication cost of 1 cycle (most of the communication is restricted to 1 cycle in this interconnect). Hence, for this interconnect, *ldst slice* performs better than *2-level*.

V. Experimental Results for Hierarchical Schemes

A. Performance Results

Figure 11 shows the results obtained with the hierarchical configurations presented in Section 4. For a 4-cluster processor, the *ldst slice* algorithm performs the best, hence the base case IPC is that obtained with a 4-cluster processor using the *ldst slice* algorithm on a crossbar interconnect. In Figure 11, we also compare the results obtained with hierarchical interconnects against the most scalable existing configuration – *first-fit* algorithm on a bi-directional ring. We also present the performance of a hypothetical clustered processor with an ideal 1-cycle crossbar. This hypothetical processor uses the *ldst slice* algorithm for instruction distribution, and is included to model one of the best possible scenarios with a clustered processor. In Figure 11, each benchmark has 2 sets of 4 bars. One set is for a 8-clustered processor, whereas the other is for a 12-clustered processor. Each bar in the graph gives the IPC improvement over that of the 4-clustered crossbar processor using the *ldst slice* algorithm.

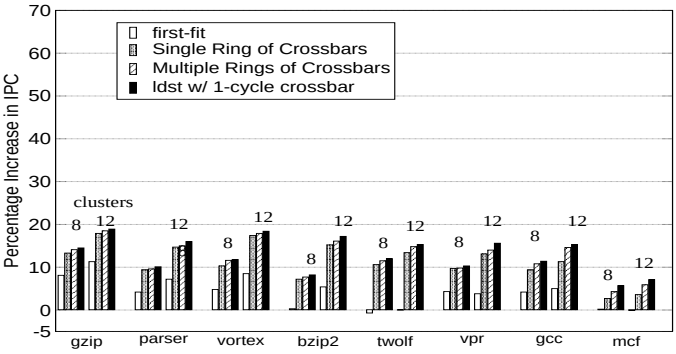

Fig. 11. Percentage IPC Increase for 8 clusters and 12 clusters, comparing Hierarchical Interconnect configurations with other configurations

From the results, we can see that performance scalability of the hierarchical approach is better than that of *first-fit* on a bi-directional ring interconnect, which has the most performance scalability of the existing algorithms, when the number of clusters is increased to 8 and 12. In fact, for `twolf`, the performance with a 8-cluster/12-cluster processor using the *first-fit* algorithm on a ring is worse than a 4-cluster processor using the *ldst slice* algorithm on a crossbar. When using the single ring of crossbar interconnect using the *2-level* instruction distribution algorithm, discussed in Section 4, for 8 clusters, there is about 10% increase in IPC over that with *first-fit* algorithm. For 12 clusters, the increase in IPC is about 15%. The corresponding values for the multiple rings of crossbars interconnect are 15% for 8 clusters and 18% for 12 clusters. It can also be seen that the multiple rings of crossbars is better than the single ring of crossbars by about 4%. *Note that the experiments were conducted without any bandwidth limitations. Hence, the performance increase seen in the figure is from low latency inter-cluster communication and not from increase in bandwidth when increasing the connectivity.*

From the graphs, it can also be seen that the performance of our hierarchical approaches is very close to that of the hypothetical 1-cycle crossbar processor. For a 12 cluster processor, the hierarchical approach's performance is only about 2% worse than that of the hypothetical 1-cycle crossbar, with a maximum performance difference of about 5%. This shows that the hierarchical approach performs very close to an ideal interconnect (with a 1-cycle communication latency).

B. Inter-Cluster Communication and Load Imbalance Analysis

The analysis is done for the single ring of crossbars configuration, a bi-directional ring using *first-fit* algorithm, and the ideal hypothetical 1-cycle crossbar using the *ldst slice* algorithm. Figure 12 presents statistics on the average number of instructions stalled in a cycle due to inter-cluster communication (left part of figure) and due to load imbalance (right part of figure). The method used to determine the values in the figure is the same as in Section 3.3. The format of this figure is as follows. The measurements are done for the 4, 8 and 12 clusters respectively for all the benchmarks. For each benchmark, there is a set of 3 bars, one for each configuration of the clusters.

The data presented in Figure 12 further clarify the IPC values presented in Figure 11. *First-fit* almost always has slightly fewer number of instructions stalled due to inter-cluster communication, but suffers from a lot of load imbalance as compared to the *hierarchical approach* and *ldst with 1-cycle crossbar*, and hence performs the worst among the three. On the other hand, the average number of instructions stalled due inter-cluster communication and load imbalance are almost the same for the *hierarchical approach* and *ldst with 1-cycle crossbar*, with the *hierarchical approach* having slightly more stalled instructions. This leads to the almost similar performance of the two approaches as seen in Figure 11, with the *hierarchical approach* performing slightly worse than *ldst with 1-cycle crossbar*.

C. Impact of Interconnection Latency

In the experiments performed so far, we considered that the communication latency between the neighboring groups of clusters, in the hierarchical interconnects, is 1 cycle. The communication latency also depends on the physical distance between the clusters, for which a detailed layout of the clustered processor is required. A layout of the processor is out of the scope of this paper. However, in this section, we study the performance impact of increasing the communication latency between the neighboring cluster groups to 2 and 3 cycles. Figure 13 presents the results of this study. The format of Figure 13 is similar to that of Figure 11. The *first-fit* configuration in Figure 13 is also the same as that in Figure 11.

Figure 13 shows that the IPC of the hierarchical interconnects reduces as the communication between the cluster groups is increased. However, even with an increased communication latency, the hierarchical interconnects are still performing better than the *first-fit on a bi-directional ring* configuration (first bar in Figure 13), because most of the communication in the hierarchical interconnect is still limited within the 1-cycle crossbar interconnect. Another important observation that can be made from Figure 13 is that the *multiple rings of crossbars* configuration has relatively more performance impact than the *single ring of crossbars* configuration, because the *multiple rings of crossbars* configuration has more communication between the cluster groups due to the distribution algorithm used.

VI. Advantages of the Hierarchical Approach

In the previous sections, we saw that the hierarchical approach is much more scalable than any of the existing approaches. The hierarchical approach also has many other advantages when considering processor design issues. This section discusses some of these advantages.

A. Resource Distribution

Even though a clustered processor has a distributed scheduler, it still has many centralized resources such as the branch prediction tables, and data caches. As the number of on-chip clusters increases, the latency to access these centralized

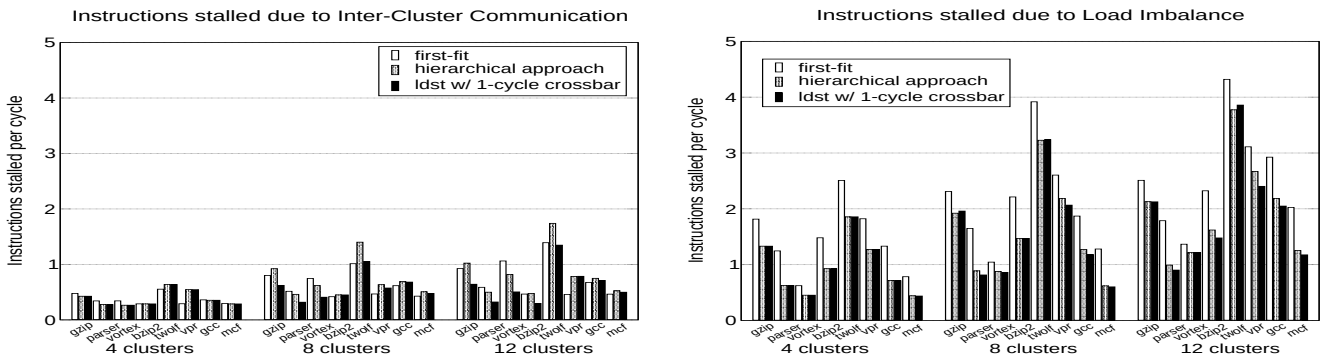

Fig. 12. Average Number of Instructions Stalled per Cycle due to Inter-Cluster Communication and due to Load Imbalance

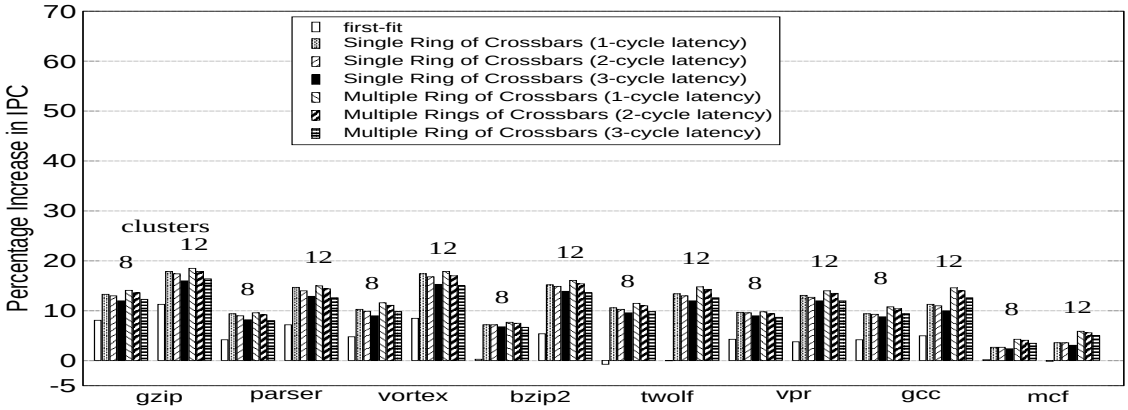

Fig. 13. Percentage IPC Increase for 8 clusters and 12 clusters, comparing Hierarchical Interconnect Configurations with other Configurations

resources may also increase (primarily as a result of the increased distance between the different resources). In this section, we focus on distributing the data cache between the different clusters. With an hierarchical interconnect, clusters are partitioned into closely knit groups, each of which can share a centralized resource within the group. For a non-hierarchical approach, on the other hand, there are no groups of clusters (each cluster operates separately), and while distributing resources, each cluster may get a separate resource. This can lead to over-distribution of resources, leading to major performance effects. For example, if a separate data cache is assigned to each cluster for a 12-cluster ring interconnect, then it takes 6 additional cycles for a cluster to access the cache local to the farthest cluster. Whereas, for a 12-cluster hierarchical interconnect with a separate cache for each 4-cluster group, access to any remote cache requires only 1 extra cycle. The distribution of the cache for an 8-cluster hierarchical single ring of crossbars interconnect is shown in Figure 14. We measure performance of a distributed data cache against a centralized cache having higher latency. These experiments are only performed for the single ring of crossbars configuration using the *2-level* algorithm for instruction distribution. The results for the multiple rings of crossbars are very similar.

For a centralized cache, the cache access latency is increased by 1 cycle when increasing the number of clusters from 4 to 8 and by 2 cycles when going from 4 clusters to 12 clusters. Each group of 4 clusters has a common centralized cache, which is called the *local cache*. Access to the local cache

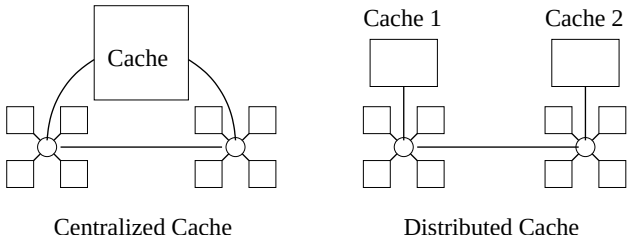

Fig. 14. Cache Partitioning Among Cluster Groups Connected by Hierarchical Interconnects

requires the same access latency as in the case of a 4-cluster crossbar processor. The accesses to the *remote cache* requires an additional cycle. This structure is somewhat similar to the NUMA architecture. In our partitioned cache configuration, we assume centralized *load/store queues* for all the partitioned caches, to effectively manage dependencies between the loads and the stores. No changes were made to the 2-level instruction distribution algorithm. We found that an alternate algorithm, that takes into consideration the distributed nature of the cache, and assign load instructions to clusters local for the cache caching the data at the load address, scattered the dependent instructions, leading to a somewhat worse performance as compared to the original algorithm.

Figure 15 gives the IPC improvement obtained when using a distributed cache, as compared to a centralized cache with increased access latency. As can be seen in the figure, a distributed cache is performing on an average 5% better than

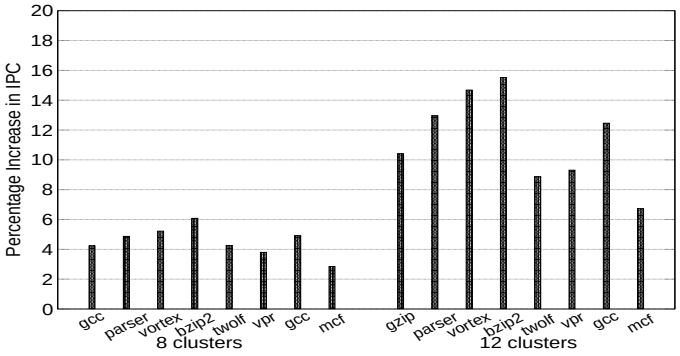

Fig. 15. Performance of distributed cache compared to centralized cache with increased latency

a centralized cache for 8 clusters, and on an average 12-13% better for 12 clusters. The maximum improvement is obtained for `bzip2`: 6% for 8 clusters and 15.5% for 12 clusters. IPC improvements are observed for a distributed cache because, for a centralized cache, all of the cache accesses incur the increased access latency, whereas for a distributed cache, only the accesses to the remote cache incur the high latency. All the accesses that are satisfied by the local cache do not incur the increased access latency.

B. Ease of Instruction Distribution

Most of the proposed clustered processor architectures employ a dispatch-time cluster assignment, except for [3]. Bhargava et. al. [3] propose a retire-time cluster assignment, by performing data-dependence analysis on traces of instructions at commit-time, and storing these traces (along with cluster assignment) in a trace cache. In dispatch-time cluster assignment, instruction distribution is done using hardware logic, which lies in the critical path of program execution. As the number of clusters increase, the logic for instruction distribution may become very complex and require extra cycles. This is because, the logic needs to consider the state of all clusters. Retire-time cluster assignment [3] is beneficial in this situation because it eliminates the critical dispatch-time latency of the cluster assignment logic. However, retire-time cluster assignment suffers from a lack of the most current information regarding the availability of operands, and can result in sub-optimal distribution, especially for large number of clusters. In hierarchical instruction distribution, only a small number of clusters are considered during distribution. It thus avoids the increase in dispatch-time cluster assignment latency (making the distribution faster), while utilizing the most current instruction operands information for cluster assignment.

We performed experiments by varying the number of cycles taken for deciding instruction placement. The comparison is done between an hierarchical interconnect using hierarchical distribution algorithm (in Section 3) and the hypothetical processor using the *ldst slice* algorithm for a 1-cycle crossbar interconnect. The time taken for instruction distribution for the hypothetical processor is increased by 1 cycle when going from 4 clusters to 8 clusters and by 2 cycles when going from 4 clusters to 12 clusters. For the hierarchical distribution, no change is made to the time taken for distribution, as the number of clusters input to the algorithm does not change. Figure 16 gives the results of the experiments.

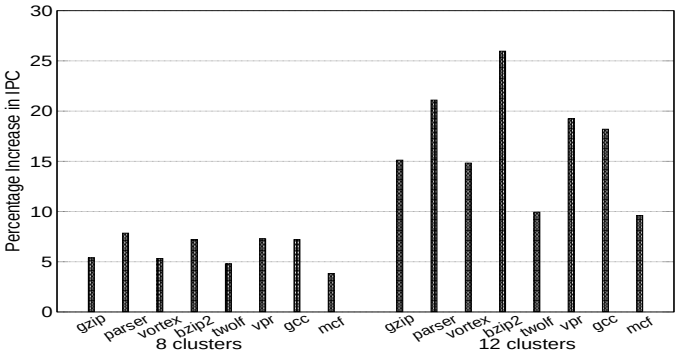

Fig. 16. IPC Improvement for the hierarchical approach as compared to the 1-cycle crossbar processor with increased instruction distribution latency

In Figure 16, Y-axis shows the percentage of the increase in IPC for the hierarchical approach over the ideal processor that requires more cycles for instruction distribution. As seen earlier in Figure 12, the hypothetical processor performed 2-3% better for 8 and 12 clusters, than the single ring of crossbars interconnect. Whereas, in Figure 16, it can seen that the hierarchical approach performs on an average 7% better for 8 clusters and 20% for 12 clusters over the hypothetical processor with increased time for distribution. The degradation in IPC for the multi-cycle instruction distribution is because the dependent instructions are not able to issue in consecutive cycles. The degradation can be reduced with a pipelined distribution logic (study of pipelined distribution is beyond the scope of this paper). Hence, if instruction distribution latency increases as the number of clusters increases, it is best to go for the hierarchical approach.

VII. Conclusions

In clustered processing models, the common goal is to decrease hardware complexity, increase clock rate, and maintain high levels of parallelism by distributing the dynamic instruction stream among several clusters. The small size of the cluster windows greatly reduces the complexity of their issue logic, allowing a higher clock rate, while the combined issue rates of several clusters still allow large amounts of parallelism to be exploited. In the long run, as more and more transistors are integrated into a processor chip, the number of clusters will increase, necessitating the development of new instruction distribution algorithms and new inter-cluster interconnects.

We first presented a detailed discussion of the effect of various hardware parameters on the relative performance of existing distribution algorithms through the stereoscope of inter-cluster communication and load balancing. We then performed a detailed study of the scalability of clustered processors with several of the existing distribution algorithms. We found that among the existing algorithms, there is no single algorithm that works consistently better for all hardware configurations. Hardware parameters such as fetch size, cluster issue width, cluster window size, and number of clusters play

a major role in deciding the relative performance of different algorithms. For a small number of clusters, algorithms that give importance to both load balancing and communication tend to perform better, whereas for larger numbers of clusters algorithms geared to reduce inter-cluster communication tend to perform better. The study showed that the existing algorithms do not scale well, as compared to a hypothetical superscalar with a single large dynamic scheduler.

To improve the scalability of clustered processors, we proposed two hierarchical interconnects and investigated techniques of distributing the instruction stream to take advantage of these new interconnects. Using this new distribution approach, we achieve performance almost equal (around 2% less) to that obtained with a hypothetical 1-cycle latency crossbar (for a large number of clusters). These hierarchical interconnects achieve IPCs around 15-20% better than the most scalable existing configuration of clustered processors.

We also discussed few of the other advantages of hierarchical interconnects in the design of on-chip clustered processors. In particular, we looked at its advantages for distributing resources among the clusters, and for achieving less complex and fast distribution hardware.

ACKNOWLEDGMENT

This work was supported by the U.S. National Science Foundation (NSF) through a CAREER grant (MIP 9702569) and a regular grant (CCR 0073582). The authors would also like to thank the reviewers for their insightful comments.

REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler and D. Burger, "Clock Rate versus IPC: the End of the Road for Conventional Microarchitectures," *Proc. ISCA-27*, 2000.

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

[3] R. Bhargava, and L. K. John, "Improving dynamic cluster assignment for clustered trace cache processors," *Proc. ISCA-30*, 2003.

[4] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," *Proc. MICRO-33*, 2000.

[5] D. Burger and J. R. Goodman, "Guest Editors Introduction: Billion-Transistor Architectures," *COMPUTER*, Vol. 30, No. 9, pp. 46-49, September 1997.

[6] R. Canal, J. M. Parcerisa and A. Gonzlez "Dynamic Cluster Assignment Mechanisms," *Proc. HPCA-6*, 2000.

[7] R. Canal, J. M. Parcerisa and A. Gonzlez "Dynamic Code Partitioning for Clustered Architectures," *International Journal of Parallel Programming*, 2000.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proc. Micro-30*, 1997.

[9] K. Hwang, "Advanced Computer Architecture," *McGraw-Hill*, 1992.

[10] G. A. Kemp and M. Franklin, "PEWS: A Decentralized Dynamic Scheduler for ILP Processing," *Proc. International Conference on Parallel Processing (ICPP)*, Vol. I, pp. 239-246, 1996.

[11] K. Kailas, K. Ebcioglu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors," *Proc. HPCA-7*, 2001.

[12] D. Leibholz and R. Razdan, "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," *Proc. Compcon*, pp. 28-36, 1997.

[13] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," *Proc. ISCA-24*, 1997.

[14] J. M. Parcerisa and A. Gonzalez, "Reducing Wire Delay Penalty through Value Prediction," *Proc. Micro-33*, 2000.

[15] J. M. Parcerisa, J. Sahuquillo, A. Gonzalez and J. Duato, "Effi cient Interconnects for Clustered Microarchitectures," *Proc. PACT-11*, 2002.

[16] N. Rangananathan and M. Franklin, "An Empirical Study of Decentralized ILP Execution Models," *Proc. ASPLOS-VIII*, 1998.

[17] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. Micro-29*, 1996.

[18] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," *Proc. Micro-30*, 1997.

[19] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *Proc. ISCA-22*, 1995.

[20] S. S. Sastry, S. Palacharla, and J. E. Smith, "Exploiting Idle Floating-Point Resources For Integer Execution," *Proc. PLDI*, 1998.

[21] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, pp. 28-40, April 1996.

[22] The National Technology Roadmap for Semiconductors, Semiconductor Industry Association, 1999.

Aneesh Aggarwal received the ME degree in Computer Science and Engineering from Indian Institute of Science, Bangalore in 1999, and the PhD degree in Electrical Engineering from University of Maryland, College Park in 2003. He is currently a faculty member of the department of Electrical and Computer Engineering at Binghamton University, New York. His research interests lie in the area of computer systems.

Manoj Franklin is a faculty member of the Department of Electrical and Computer Engineering at University of Maryland, College Park. He received his B.Sc. (Engg) in Electronics and Communications from the University of Kerala, and the M.S. and Ph.D. degrees in Computer Science from the University of Wisconsin-Madison. Prior to joining University of Maryland, he was a faculty member of the Electrical and Computer Engineering Department at Clemson University. His research interests are within the broad area of Computer Architecture.