A Communication-Centric Approach To Instruction Steering For Future Clustered Processors

Jason Cong, Ashok Jagannathan, Glenn Reinman, Yuval Tamir
Computer Science Department
University of California, Los Angeles
{cong,ashokj,reinman,tamir}@cs.ucla.edu

Abstract

In any clustered processor, there is a mechanism that determines the assignment of instructions to clusters. One key goal of this instruction steering mechanism is to minimize inter-cluster communication. As feature size decreases and the number of clusters increases, the relative cost of inter-cluster communication will increase, requiring even greater focus on optimizing inter-cluster communication. To this end, we propose the CSPAN (Communication Span) instruction steering algorithm that controls the communication cost by controlling the distance that values must traverse to reach dependent instructions. This is done without imposing absolute constraints on resource utilization. The scheme dynamically adapts to the characteristics of the program, imposing communication locality only to the extent that it is beneficial to performance.

Experimental results demonstrate that CSPAN is able to outperform the best known Advanced RMBS steering algorithm by around 13% over a set of SPEC2000 and MediaBench applications. This improvement comes from a reduction of 28% in the inter-cluster communication penalty without any noticeable change in resource contention. Also, CSPAN is able to improve performance by 10% compared to prior work on dynamic communication/parallelism tradeoff.

1 Introduction

Multicluster architectures [1, 2, 3, 4] have been proposed as a way to address interconnect, power, and thermal problems which are of primary concern in future technologies. A crucial component of clustered processors is the instruction steering algorithm, and a number of studies [5, 6, 7, 8] have looked at different steering heuristics that attempt to maximize performance by balancing two conflicting constraints: *resource utilization* and *inter-cluster communication*. While several algorithms have been proposed for instruction steering, the Advanced RMBS algorithm by Gonzalez, et al. [6] and its variants [7, 8] were shown to produce the best results under different cluster interconnection topologies. This algorithm attempts to dynamically balance resource utilization and communication penalty by sending instructions to clusters where the inputs are produced, unless a workload imbalance among the clusters is detected; in this case the instruction is steered to the least loaded cluster. As the number of on-chip clusters increases [9] and wire delays start to dominate the performance [10], instruction steering algorithms for clustered processors have to become more communication-centric as concluded in the study by Franklin, et al. [11].

In order to motivate an interest in communication-centric steering algorithms, we show performance data for a few SPEC2000 and MediaBench [12] benchmarks in Figure 1. These results were obtained on a processor with 16 clusters[1] with a ring interconnect. The first three bars show the IPC when using 4, 8, or 16 of the total available clusters. In order to separate the impact of increased resources alone, the last two bars show the IPC when using 8 or 16 clusters, assuming zero inter-cluster communication penalty.

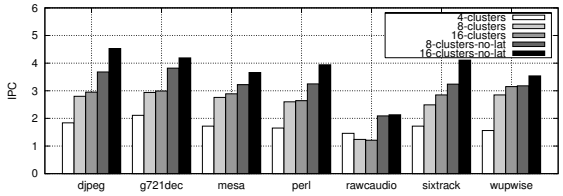

Figure 1: Performance of the Advanced RMBS instruction steering algorithm on a 16-cluster processor with a ring interconnection.

We make two observations from Figure 1: (1) Bars 1,2 and 3 show that several programs are able to derive benefit with increased resources. However, in the presence of inter-cluster communication penalty, the Advanced RMBS algorithm is not able to use additional resources

[1] Processor parameters are presented in Section 4

(beyond 8 clusters) effectively. This is seen from Figure 1 where there is a marginal improvement in performance going from 8 to 16 clusters (bars 2 and 3), while there is a larger performance gap due to increased resources (as shown by bars 4 and 5). In the case of *rawcaudio*, the performance even drops after 4 clusters, which is consistent with the observation made by Balasubramonian, et al. [9]. (2) Comparing bars 2,3,4 and 5, we see that there is a large gap in performance with and without communication penalty. This indicates that if we can directly control communication penalty without limiting resources, there is potential for improving performance by using available resources more efficiently.

While the performance of the Advanced RMBS algorithm with increased resources (as shown in Figure 1) can be attributed to several factors such as lack of ILP or incoherent program behavior, we show that using a single *workload balance factor* limits the ability to effectively decouple resource utilization and inter-cluster communication when the number of clusters increases. This workload balance factor tightly couples resource utilization and inter-cluster communication and is not able to control one independent of the other. Ideally, we would like to use as much of the available resources as possible to speed up the program as long as we do not increase the average inter-cluster communication latency seen by the program execution.

Based on this observation, we propose **CSPAN**, a steering algorithm which attempts to directly control the average inter-cluster communication latency without restricting resource utilization significantly. The basic idea of our steering algorithm is to limit the steering of instructions to only those clusters physically close to where the inputs are produced, and the number of neighbors to consider is dynamically adjusted depending on the program's tolerance to inter-cluster communication penalty. Experimental results show that CSPAN is able to outperform the Advanced RMBS algorithm by 13% on the average for several SPEC2000 and MediaBench applications, and the performance difference increases to 19% when the inter-cluster communication latency is doubled. We also show that the effective decoupling of resource utilization and communication penalty enables CSPAN to improve performance by 10% compared to a resource-limiting approach [9] to dynamic communication/parallelism tradeoff.

The remainder of the paper is organized as follows: The baseline processor model is presented in Section 2. A dynamic instruction steering algorithm, CSPAN, is discussed in Section 3. Section 4 presents detailed experimental results obtained with CSPAN and studies the sensitivity of the steering to communication latency and interconnection topology. Related work is presented in Section 5 and the paper concludes with future directions in Section 6.

2 Baseline Clustered Processor

Our baseline multicluster architecture is based on the work by Zyuban [13], and a processor with two clusters is shown in Figure 2. Each cluster is provided with a local issue window, register file, and a memory disambiguation unit. The front-end uses a combination of BBTB [14] and g-share [15] based branch predictor to supply instructions to the execution core. Once instructions are fetched, they are decoded and allocated space in the ROB. Memory operations need to be allocated space in the Load/Store window, but this is done only after a decision regarding cluster assignment is made. Instruction steering in our baseline processor is based on the Advanced RMBS approach [6], which schedules an instruction to the cluster that produces its input operands, unless there is a large imbalance in the load on other clusters. The workload balance is measured using the DCOUNT [7] metric.

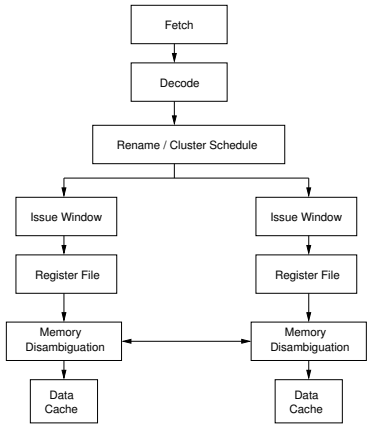

Figure 2: Baseline clustered architecture

The register file in our baseline architecture is *split* among different clusters and only a subset of the physical registers is accessible from a given cluster. This requires that register data be communicated between dependent instructions scheduled to different clusters, and we use the remote register communication methodology proposed in [13] for this purpose. Also, our Load/Store window is split among the clusters to enable distributed memory disambiguation.

We examined different cache organizations studied earlier [13, 16] and found that replicating the L1 data cache banks at each cluster produced better performance than an interleaved cache when the communication latency increases (as with a 16-cluster processor). Since the cache

banks are replicated, there is no restriction on where memory operations can be scheduled. Each memory operation is decoded into two micro-operations: an address calculation operation and a memory access operation. In order to preserve correct memory ordering among instructions scheduled to different clusters, a store operation is allocated space in the Load/Store window of every cluster. Store address and data are broadcast to all clusters so that that each memory disambiguation unit can make decisions locally. A load instruction is only allocated space in the cluster to which it is steered by the cluster scheduler. Within each Load/Store window, we speculatively assume that a load is dependent upon all the prior stores whose effective addresses are unknown. A load operation is allowed to execute when it is known not to alias any of the prior store instructions in the window. If it does, the load is delayed until the store data is available, so that the data can be forwarded locally from the Load/Store window.

Interconnect Model

A clustered processor with a large number of clusters can be laid out in a variety of ways with different interconnection topologies. The actual delay between the clusters depends on the layout and the assumptions on the interconnection structure. For most part of this work, we focus on a ring interconnection model, primarily due to its low complexity. We assume that separate networks exist for register value communication and memory data communication, similar to the assumptions in [9].

Fetch	4K entry 4-way associative BBTB 32KB g-share
I-cache	64KB, 2-way, 2 ports
Clusters	16
Interconnect	Ring
Decode Width	8
Issue Width	1 per cluster
Issue Window	16 entries per cluster
ROB	512 entries
LSQ	128 entries total
Register File	72 entries per cluster
Functional Units	INT – 1 per cluster FP – 1 per cluster
D-cache	Replicated 16KB, 4-way, 32 bytes/block 1RW port per cluster 1 cycle latency for access multi-cycle store address/data broadcast
Unified L2	1MB, 8-way, 64 bytes/block 12 cycle latency Shared bus to DL1
Main memory	180 cycles for first chunk

Table 1: Baseline processor parameters used in this study

Under this interconnection model, each cluster is directly connected only to its neighbors, and communication between non-neighboring clusters takes multiple cycles depending on the assumptions on inter-cluster communication latency. Based on the technology parameters for interconnects, delay of the ALU and the number of functional units per cluster, we set our cycle time to accommodate single-cycle bypassing within the same cluster. Two unidirectional rings are assumed to exist for communication in both directions, so that on a N cluster machine, a total of $2 \times N$ data transfers can be done in a cycle. The baseline processor parameters used in our experiments are shown in Table 1. Latency values for the caches were obtained using CACTI [17] for a futuristic 70nm process technology.

3 CSPAN: Communication-Centric Instruction Steering

As resources are partitioned and physically distributed in a clustered processor [1, 3], instruction steering algorithms have to carefully balance inter-cluster communication latency with resource utilization in order to maximize performance. While several techniques [6, 5, 7, 8] have been explored in this regard, the Advanced RMBS [2] [6] algorithm (and its variants [7, 8]) proposed by Gonzalez, et al., was shown to outperform other heuristics. In Figure 3, we briefly review the AdvRMBS steering algorithm used in this work, which is a combination of the techniques in [6, 7].[3] This algorithm dynamically balances

If workload imbalance is higher than a *threshold*,
  send to the *least loaded* cluster.
Else,
  If all inputs are ready or all are being produced,
    send to the *least loaded* among the ones where
    the inputs are mapped.
  Else,
    send to the cluster where an input is being produced,
    so that communication penalty of transferring the
    produced input can be overlapped with the computation.

Figure 3: Description of the Advanced RMBS steering algorithm.

resource utilization and inter-cluster communication by using a single workload balancing factor to trade off communication and computation. Though it is possible to use different metrics to capture workload imbalance, we use the DCOUNT [7] metric, which was shown to produce better performance. Essentially, this metric uses the difference in the total number of instructions dispatched to a specific cluster and the average number of instructions dispatched per cluster to detect imbalance in cluster workload.

When the number of clusters is large, sending instructions to the least loaded cluster independent of its physical proximity to where the instruction's inputs are produced leads to an increased communication penalty, and hence decreased performance. This observation is consis-

[2] Henceforth referred to as AdvRMBS.

[3] We do not use the topology-aware RMBS [8] here as we do not have multiple copies of a register value available in different clusters.

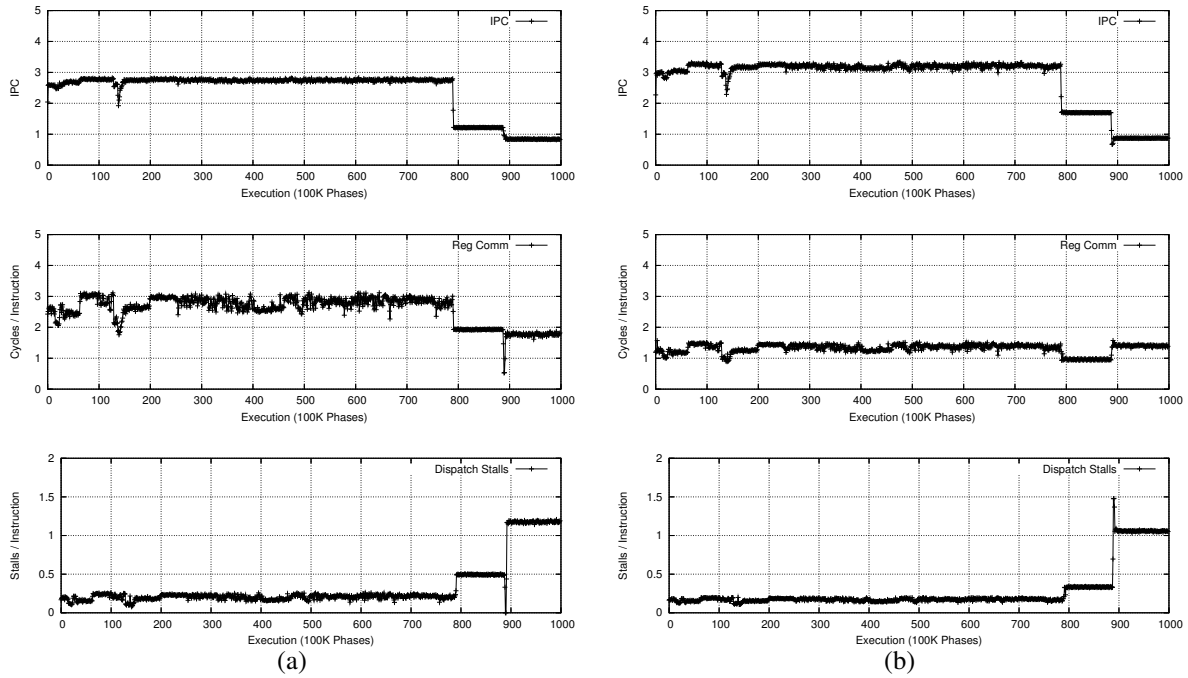

Figure 4: IPC, register communication (RegComm) and dispatch stalls (Dispatch Stalls) collected during the execution of SPEC benchmark `bzip2` on a 16-cluster processor with ring interconnect using (a) AdvRMBS for instruction steering, with span 8 and (b) CPSAN for instruction steering, with fixed span 1.

tent with the study by Franklin, et al., [11] which showed that dependence-based instruction steering scheme with workload balancing does not scale well with an increasing number of clusters as it places mutually dependent instructions in far-apart clusters.

In order to overcome this limitation, we propose to restrict the instruction steering algorithm by considering physical proximity when steering an instruction to a cluster that is different from where its inputs are produced. We define the *span* of the steering algorithm as a limit on the distance (in hops) to the neighboring clusters to consider when trying to steer around a *desired* cluster. The desired cluster is always chosen based on the input dependencies for the instruction. For example, with a ring interconnect, a span of 1 would only allow the steering algorithm to consider the two adjacent clusters to detect workload imbalance, and to steer the instruction to in case of an imbalance. This restriction ensures that an instruction is *always* steered close to one of the clusters where the inputs are mapped to.

We call this approach CSPAN (for *communication span*), as it explicitly controls the communication penalty incurred during instruction execution. In comparison, the span of the AdvRMBS algorithm for a 16-cluster processor with a ring topology is 8, as the algorithm considers all clusters when it is trying to steer an instruction around the desired input-producing cluster.

To illustrate the effect of CSPAN instruction steering, we collected three statistics: IPC, average register communication per instruction and average number of dispatch stalls per instruction for two scenarios on a 16-cluster processor connected with a ring interconnect using (a) AdvRMBS algorithm for instruction steering, where the span is 8, and (b) CSPAN for instruction steering, with a fixed span of 1. An instruction is considered to be stalled during dispatch if there is no space in the issue window of the chosen cluster. The register communication latency for an instruction is the number of cycles the instruction waits in the issue window before the input operand(s) arrive once it is produced. The workload balancing threshold that produced the best performance was chosen for each steering algorithm. These statistics were collected for intervals of 100K instructions over several million instructions for the SPEC benchmark `bzip2`, after fast-forwarding through the initialization portion of the code based on [18]. These results are shown in Figure 4.

As the physical locality restriction on steering always ensures that an instruction is steered close to at least one of its inputs, Figure 4(b) shows that this is able to reduce the average communication latency by almost 50% during most of the execution compared to AdvRMBS steering in

Figure 4(a). One would expect CSPAN to incur more dispatch stalls due to the restriction on resource utilization. However, in this particular case, the reduction in inter-cluster communication decreases the time an instruction spends in the issue window, and this helps to offset the restriction on resource utilization. Thus, either the dispatch stalls remain almost the same (as during the first 80M instructions) or are lesser compared to the AdvRMBS steering in Figure 4(a). These two trends combined can improve the IPC for this application by around 15% for this execution phase.

The reason why even the best workload balancing threshold in Figure 4(a) could not produce the trend in Figure 4(b) is because of the tight coupling between workload balancing and communication penalty, the factors considered by the AdvRMBS algorithm. A high workload balancing factor tends to distribute instructions on all clusters, increasing the communication penalty, while a low workload balance factor makes less use of available resources (by sending instructions mostly based on input dependencies) while reducing the communication penalty significantly. However, the restricted CSPAN steering in Figure 4(b) attempts to decouple resource utilization from communication penalty. This allows the communication penalty to be reduced without significantly limiting the available resources, and hence translates into an improvement in performance.

If both inputs are ready or both are being produced
  If workload imbalance among clusters in $T1$ or T_2 or both,
    send to least loaded cluster in T_1 or T_2
    or $T_1 \cup T_2$ respectively.
  Else send to c_1 or c_2, whichever is least loaded.
Else (W.L.O.G., assume i_1 is ready and i_2 is being produced)
  If workload imbalance in T_2,
    send to the least loaded cluster in T_2.
  Else send to c_2.

Figure 5: Description of CSPAN steering algorithm.

For completeness, we summarize the CSPAN steering algorithm in Figure 5 using the following notations: i_1 and i_2 are the two inputs (at most) for the instruction, and c_1 and c_2 are the clusters where these inputs are produced respectively. S is the current span of the steering algorithm, T_1 is the set of clusters within distance S from $c1$, and T_2 is the set of clusters within distance S from c_2. Instruction dispatch is stalled if the chosen cluster is full in any of the above cases.

Though it is possible to statically fix the span to some value that works well for most applications, prior work [9] shows that application demands change over time, and it is beneficial to dynamically manage the trade-off between computation and communication. Consequently, we present a technique to dynamically adjust the span of the steering algorithm based on the application's behavior. Intuitively, the span has to be smaller during communication-bound program phases and can be larger when there is more resource requirement and tolerance to inter-cluster communication latency.

Dynamically Varying the Span

A large body of research [19, 20, 21, 9] exists on dynamically tuning the hardware depending the application requirements. Many of these techniques are interval based in that they use statistics collected about the dynamic behavior of the program in the past few intervals to guide the hardware tuning process. Our approach also uses an interval-based scheme where we start with a span of one at the beginning of program execution and gradually increase or decrease the span by one depending on the impact on the performance. We define an *interval* as a consecutive sequence of dynamic instructions and a program phase as a consecutive sequence of intervals with IPC variation less than a predefined *IPC_threshold*. We detect the start of a new phase if the measured IPC for a small sequence of *sampling_interval_length* intervals does not vary beyond the *IPC_threshold*. We also use a *saturation_interval_length* counter to periodically force an evaluation of span change to ensure that optimization opportunities are not missed during lengthy program phases. This counter is initialized at the start of a phase, and reset each time it saturates. To dynamically change the span, we distinguish the following two scenarios:

1. We intentionally change the span to study the effect on program behavior. This is the case during the start of the program, or when the *saturation_interval_length* counter saturates during a program phase.

2. There is a change in the IPC of the program beyond the *IPC_threshold* and the span has to be adjusted accordingly. This happens when the program is transitioning from one phase to another or is going through an unstable phase.

Assume that the steering algorithm is currently using a span S. For scenario (1) we simply probe for the span that produces the the best stable performance, where stability is measured over *sampling_interval_length* intervals. The reference IPC used for comparison is the IPC of the first interval at the start of the program or the phase. We probe by first evaluating if changing the span to $S + 1$ can improve the performance beyond *IPC_threshold*. If it does, we register the modified span, consider this as a start of a new phase and record the reference IPC, and continue to evaluate if increasing the span will again help. We stop this probing of larger span values when either

the performance degrades or does not improve beyond the *IPC_threshold* with respect to the current reference IPC or we hit the maximum span value, which is the maximum distance between any two clusters. At the end of this process, if the span has increased from S, we stop probing here. However, if incrementing the span never helped, we start to probe smaller span values from $S-1$ in a similar fashion until we find the span which produces the best IPC, or we hit the minimum span value of one. If neither incrementing nor decrementing the span improved the performance, we continue execution with a span S until the next opportunity for changing the span.

In order to handle scenario (2), it is instructive to look at Figure 4 which shows that a change in the IPC is often associated with a change in one or both of the two conflicting parameters in clustered processors: the average inter-cluster communication penalty (Reg Comm) and the average resource utilization constraint (Dispatch Stalls). Such a correlation among different program parameters during different execution phases was also observed in [18]. Over multiple simulations, we observed that an *increase* in IPC is typically associated with a decrease in dispatch stalls or register communication or both. Similarly, a *decrease* in IPC is associated with an increase in either or both of these factors. If we can precisely identify the dependency between IPC changes and these two factors, we can take appropriate action: increase the span if the application is resource bound, or decrease the span if it is more communication bound.

To identify significant changes in the register communication penalty and dispatch stalls, and to relate it to the changes in IPC, we maintain two threshold values: *reg_comm_threshold* and *dispatch_stall_threshold*. If any of the two factors change beyond their respective threshold values, it is considered significant enough to impact the IPC. For example, if an increase in IPC comes only from a reduction in dispatch stalls, we see this as a potential to improve the IPC by reducing the dispatch stalls further by increasing the span. Thus, in this case, the heuristic only attempts to increment the span. However, when we cannot exactly identify which of the two parameters impacted the IPC (when both parameters change above or below their threshold values), we probe for the best span similar to what we do for scenario (1) above.

4 Experimental Results

In this section we demonstrate the performance improvement obtained by using CSPAN to steer instructions in a highly clustered processor. The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [22], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. We performed experiments on 23 SPEC2000 applications and 13 MediaBench applications, 36 in total, which we could get working with our simulator. We used the SimPoint [18] tool to fast-forward all the SPEC benchmarks over the initialization phase and simulate for 100M instructions to measure the statistics. All MediaBench programs were run to completion as the total number of instructions was less than one billion.

Simulation results showed that CSPAN improved the performance over AdvRMBS in almost all the cases. In summary, there were 22 programs with more than 5% improvement, 13 programs with less than 5% improvement and 1 program, *lucas* where the performance decreased by 7%. We observed that the degradation for *lucas* is due to the limitation of our span changing heuristic. As we use a 5% variation in IPC to decide whether the span has to be changed, *lucas* got stuck at a situation where a larger than 5% change is generated only when the span is changed immediately by more than one. Since our heuristic changes the span only by 1 each time, it did not accommodate this program behavior of *lucas* during the final stage of execution.

For brevity reasons, we present results on 15 of these programs: four SPEC2000 INT, four SPEC2000 FP, and seven MediaBench, which include low and high IPC codes. These programs were chosen to demonstrate the behavior of CSPAN under different communication and resource utilization constraints, and to emphasize the robustness of the technique under different workloads.

Figure 6 summarizes the performance of the various programs executing on the baseline processor with 16 clusters and using two different steering schemes: AdvRMBS and CSPAN. For comparison purposes, we show three data points for the Advanced RMBS approach: AdvRMBS4, AdvRMBS8 and AdbRMBS16, which correspond to using 4,8 and 16 of the available clusters respectively. These results were obtained by determining the best workload balance factor for AdvRMBS for each configuration, and using the same value for all programs for that configuration.

Figure 6 shows that the CSPAN approach can consistently outperform the AdvRMBS approach in all the cases. The harmonic mean shows that the CSPAN is 47% better than AdvRMBS4 and around 13% better than both AdvRMBS8 and AdvRMBS16. Comparing CSPAN and AdvRMBS16, significant improvements in IPC are seen for the MediaBench applications *g721dec*(24%), *g721enc*(26%) and *rawdaudio*(30%) while several other benchmarks see improvements of over 10%.

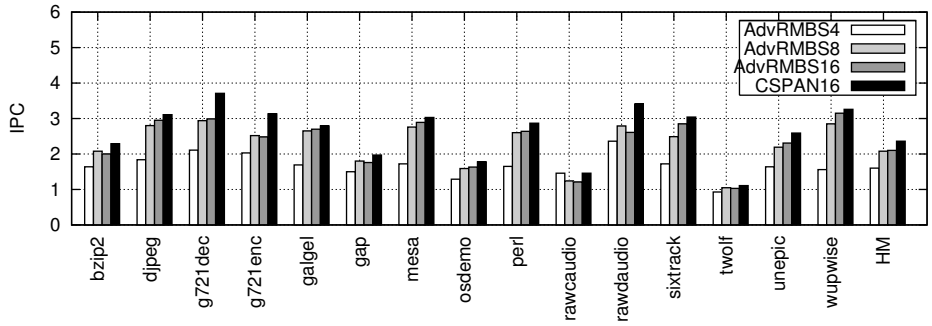

Figure 6: Comparison of IPC for different steering heuristics on the baseline processor with 16 clusters.

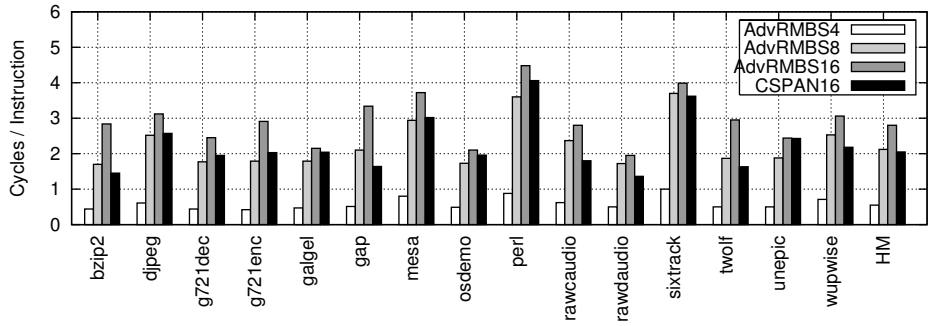

Figure 7: Average register communication latency for different steering heuristics on the baseline processor with 16 clusters.

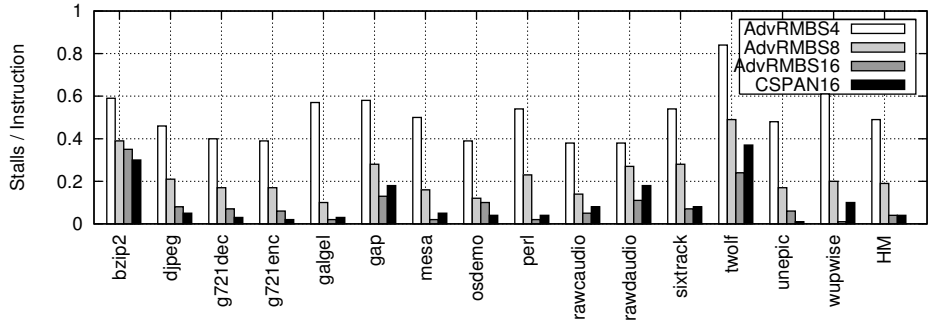

Figure 8: Average dispatch stalls for different steering heuristics on the baseline processor with 16 clusters.

The media applications which see a large improvement are audio/speech compression and decompression algorithms which execute a loop body for a significant portion of their runtime. For example, *rawdaudio* has a single variable feeding two independent streams of computations inside this loop body, with limited communication between loop iterations. Each computation path has a reasonable amount of ILP, and the CSPAN approach is able to spread out the computation while limiting the communication between the dependent instructions. Thus, it is able to make use of the available issue bandwidth more effectively to produce a larger performance difference.

To provide a better understanding of the behavior of the steering algorithms, Figures 7 and 8 show the average register communication latency and the dispatch stalls for both steering algorithms. As the CSPAN approach is able to keep the communication penalty similar to AdvRMBS8 while having dispatch stalls comparable to AdvRMBS16, it is able to achieve a better performance overall. Figures 7 and 8 show that the CSPAN approach is able to reduce the average inter-cluster communication latency by 28% compared to AdvRMBS16 without really impacting resource utilization, and this translates into a good improvement in performance.

Looking at specific instances, applications such as *bzip2*, *g721enc*, *g721dec*, and *djpeg* are able to gain from a reduction in both the inter-cluster communication and dispatch stalls. For these applications, as mentioned earlier,

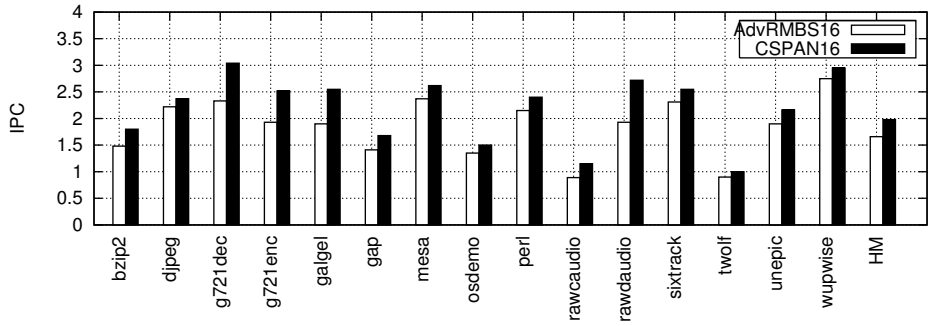

Figure 9: IPC for AdvRMBS and CSPAN for a 16-cluster processor and ring interconnection with a 2-cycle latency to communicate with neighboring clusters.

the reduction in inter-cluster communication reduces the time an instruction waits in the issue window, and this leads to a reduction in the dispatch stalls. Thus, these applications tend to see a high benefit from the CSPAN approach.

For programs such as *wupwise* and *twolf*, there is a large increase in the dispatch stalls in spite of the reduction in the inter-cluster communication. Thus, they are able to derive only limited benefit in IPC (4% and 8% respectively) from the reduction in the communication penalty induced by CSPAN. Other programs such as *gap* and *rawdaudio* are able to sustain the increase in dispatch stalls and are able to translate the reduction in communication penalty into an improvement in IPC (12% and 30% respectively).

Finally, programs such as *unepic* do not have a significant change in the communication penalty, while there is a good reduction in dispatch stalls. As CSPAN directly controls communication, more instructions tend to have register communication penalty closer to the average penalty. This reduces the instruction waiting time in the issue window, which decreases dispatch stalls and increases resource utilization. Thus, CSPAN improves the performance by around 12% in this case. This further emphasizes the advantage of directly controlling the communication penalty.

4.1 Sensitivity Analysis

Though there are several factors that influence the performance of any dynamic scheme such as CSPAN, we attempt to present the sensitivity of CSPAN to two of the most important factors: *inter-cluster communication latency* and *interconnection topology*.

Figure 9 shows the performance improvement obtained using CSPAN over AdvRMBS when the inter-cluster communication latency is increased from 1 to 2 cycles. For brevity, we only show results for AdvRMBS16 and CSPAN. CSPAN is able to improve the performance in all cases, with a maximum improvement of 41% for the program *rawdaudio* and around 30% improvement for several other programs such as *g721dec, g721enc, galgel*, and *rawcaudio*. Overall, the improvement seen is 19% as shown by the harmonic mean. These gains are obtained through a reduction of 34% in the average inter-cluster communication penalty, with a negligible decrease of 0.5% in the average number of dispatch stalls.

We also evaluated the sensitivity of the CSPAN approach to the interconnection structure. For a **MESH** interconnect where each cluster has four neighbors, the performance improvement of CSPAN over AdvRMBS reduced to around 6% on the average, with a maximum performance improvement of 12% for the benchmark *rawdaudio*. This is because of the availability of more resources with less penalty for the AdvRMBS approach, and hence the least loaded cluster selection did not significantly affect the performance for the AdvRMBS technique. Overall, the technique is quite robust to variations in some of the fundamental architectural parameters.

4.2 Communication/Parallelism Trade-off

Recent work by Balasubramonian et al. [9] showed that increasing the number of clusters degraded the performance for some programs due to dominating inter-cluster communication penalty. The authors proposed a methodology to allocate resources dynamically based on the amount of parallelism in the application so that the impact of inter-cluster communication penalty is reduced. During low ILP program phases, this technique limits the execution to the first few clusters in an attempt to control communication penalty impact. For reference, we use the term RSPAN (Resource Span) to represent an approach as in [9] which limits communication by limiting the amount of available resources. In comparison, CSPAN attempts to directly limit communication without impacting resource utilization. Since these approaches are similar in spirit (dynamically manage the communication/parallelism trade-off), this section presents a direct

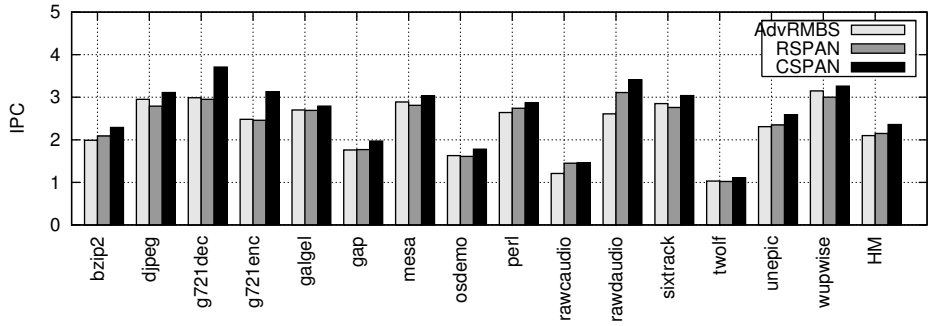

Figure 10: Comparison between CSPAN and prior work on dynamic communication/ computation trade-off.

performance comparison between them.

Notice that RSPAN can be trivially simulated with our heuristic by using the span to steer instructions only to clusters at a span distance away from cluster 0, where the program is initially steered for execution. While the CSPAN will use the span to only limit steering around where inputs are produced, RSPAN would steer instructions around cluster 0 independent of where the inputs to the instruction are produced. Instead of attempting to accurately reproduce the dynamic trade-off technique in [9], we compute an "optimum" performance for RSPAN using a method similar to the one in [23]. Essentially, we try to find the best span for each interval for a program by simulating each interval for different span values and choosing the span which gives the best performance. We do this assuming that the best performance for an interval for a span N is obtained when all the prior intervals execute at the same span. Though this ignores the effect of change in communication penalty and resource utilization due to change in span, we believe that this will only marginally impact performance for the first few hundred instructions in any interval.

Once we obtain the best span to use for each interval, we run simulations once for each benchmark and use the predetermined best span for each interval to determine the overall program performance using RSPAN. We present results comparing CSPAN with RSPAN in Figure 10 for the baseline processor with 16 clusters. For comparison, we also show the performance obtained using AdvRMBS16. Figure 10 shows that the RSPAN approach is able to match the performance of the best static organization as shown in [9]. The maximum performance improvement using RSPAN is around 20% for *rawdaudio*, while the average improvement is around 3%. However, in several cases, there is a small degradation in the performance. This is because when the span is reduced, some instructions have to be steered to a cluster where none of the instruction's input operands are produced, incurring extra communication penalty.

The CSPAN approach is able to improve performance by 10% over RSPAN on the average. This is due to several reasons. First, the RSPAN approach limits the communication penalty by limiting available resources, much similar to the AdvRMBS approach, while CSPAN is able to control communication latency without limiting resources. Second, while the CSPAN approach guarantees that an instruction is always steered closer to at least one of its inputs, the RSPAN does not have any such guarantees. Thus, it is possible to incur more communication penalty for RSPAN than CSPAN.

5 Related Work

Prior research [1, 13, 5, 6] has explored in detail various characteristics of clustered processors. These include a study of resource allocation techniques across clusters [13] including the first level data cache [16], dynamic instruction steering heuristics, and cross-cluster communication topology. Much of the prior research was conducted on small number of clusters (typically 4), and the AdvRMBS algorithm does extremely well under such circumstances. Franklin, et al. [11] studied several instruction distribution algorithms for processors with up to 12 clusters and showed that different heuristics perform differently based on the interconnection topology. Zyuban [13] also explored clustered processors in the context of energy and concluded that clustered processors are a viable alternative for the energy efficiency needs of future processors.

Related work [21, 20, 19] has looked at using dynamic program behavior information to improve performance or save energy. Most of these techniques are based on the premise that past program behavior can be used to predict future program needs. They are interval-based, in that they periodically collect relevant statistics about program behavior and apply it to efficiently use the available resources.

Balasubramonian, et al. [9] studied processors with large number of clusters and determined that it is useful to dy-

namically manage the resources and communication overhead based on application requirements. Our work is similar to theirs, except that CSPAN provides finer control over the communication/resource trade-off. Also, the CSPAN approach attempts to reduce to inter-cluster communication penalty without impacting resource utilization.

6 Summary

This paper shows that, due to on-chip wire delay, future clustered processors should use communication-aware instruction steering heuristics instead of simply trying to balance the load across all available computation resources. Specifically, we presented CSPAN, a communication-centric approach to instruction steering, and a heuristic to dynamically control the span of the steering and provide fine-grain computation vs. communication trade off. Experimental results showed that CSPAN improved performance over Advanced RMBS approach by around 13% on the average for several SPEC2000 and MediaBench programs. The improvement of 19% in performance using CSPAN with increased inter-cluster communication latency further emphasizes the importance of communication-centric instruction steering.

We believe that an approach such as CSPAN is better suited for a multithreaded environment where multiple programs are competing to use the available resources, and communication penalty has to be managed more effectively. Also, a combination of approaches such as RSPAN, which explicitly limits resources, and CSPAN, which attempts to decouple communication directly, appears to be a promising approach to performance improvement in communication dominated clustered architectures. Both these appear to be promising directions for future research.

References

[1] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," in *International Symposium on Microarchitecture*, 1997.

[2] R. Kessler, E. McLellan, and D. Webb, "The Alpha 21264 Microprosessor Architecture," in *International Conference on Computer Design*, Dec. 1998.

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[4] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal Q1*, 2001.

[5] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 337–347, 2000.

[6] R. Canal, J.-M. Parcerisa, and A. Gonzalez, "Dynamic Code Partitioning for Clustered Architectures," *Int. J. Parallel Program.*, vol. 29, no. 1, pp. 59–79, 2001.

[7] J.-M. Parcerisa and A. Gonzalez, "Reducing Wire Delay Penalty Through Value Prediction," in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pp. 317–326, 2000.

[8] J.-M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato, "Efficient Interconnects for Clustered Microarchitectures," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, p. 291, 2002.

[9] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors," in *Proceedings of the 30th Annual International Symposium on Computer architecture*, pp. 275–287, 2003.

[10] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *27th Annual International Symposium on Computer Architecture*, 2000.

[11] A. Agarwal and M. Franklin, "An Empirical Study Of The Scalability Aspects of Instruction Distribution Algorithms in Clustered Processors," in *Proceedings of ISPASS*, 2001.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 330–335, 1997.

[13] V. V. Zyuban and P. M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Transactions on Computers*, vol. 50, no. 3, pp. 268–285, 2001.

[14] T. Yeh and Y. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 129–139, Dec. 1992.

[15] S. McFarling, "Combining branch predictors," Tech. Rep. TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[16] P. Racunas and Y. N. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitectures," in *Proceedings of the 17th Annual International Conference on Supercomputing*, pp. 22–31, 2003.

[17] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," in *Technical Report*, 2001.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45–57, 2002.

[19] I. R. Bahar and S. Manne, "Power and Energy Reduction Via Pipeline Balancing," in *In Proceedings of ISCA*, 1998.

[20] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pp. 245–257, 2000.

[21] D. H. Albonesi, "Dynamic IPC/Clock rate Optimization," in *In Proceedings of ISCA*, 1998.

[22] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[23] F. Latorre, J. Gonzalez, and A. Gonzalez, "Back-end Assignment Schemes For Clustered Multithreaded Processors," in *Proceedings of the 18th Annual International Conference on Supercomputing*, pp. 316–325, 2004.