Disjoint-Path Routing: Efficient Communication for Streaming Applications

DaeHo Seo
Intel Corporation
Austin, TX, USA
daeho.seo@intel.com

Mithuna Thottethodi
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
mithuna@purdue.edu

Abstract

Streaming is emerging as an important programming model for multicores. Streaming provides an elegant way to express task decomposition and inter-task communication, while hiding laborious orchestration details such as load balancing, assignment (of stream computation to nodes) and computation/communication scheduling from the programmer. This paper develops a novel communication optimization for streaming applications based on the observation that streaming computations typically involve large, systematic data transfers between known communicating pairs of nodes over extended periods of time. From the above observation, we advocate a family of routing algorithms that expend some over overheads to compute disjoint paths for stream communication. Disjoint-path routing is an attractive design point because (a) the overheads of discovering disjoint paths are amortized over large periods of time and (b) the benefits of disjoint path routing are significant for bandwidth-sensitive streaming applications. We develop one instance of disjoint-path routing called tentacle routing – a backtracking, best-effort technique. On a 4x4 (6x6) system, tentacle routing results in 55% (84%) and 28% (41%) mean throughput improvement for high-network-contention streaming applications, and for all streaming applications, respectively.

1. Introduction

Stream-based programming models provide a natural way to express the parallelism of several media processing applications and other applications with simple forms of data- and task-parallelism. The advantages of streaming programming models have been demonstrated on research machines (MIT Raw [15], Stanford Imagine [11]) as well as on real multicore systems and accelerator-computing systems [1], [13], [2], [18], [4]. A stream program may be expressed as a graph with independent threads of computation – "actors" – at vertices and directed edges representing "communication streams" between actors. Each "actor" is an independent execution context with its own private memory that repeatedly operates on a stream of input data and emits output streams that other "downstream" actors may further act upon. Thus, the programming model enables programmers to specify task decomposition and inter-task communication while hiding orchestration details (e.g., load balancing, assignment of actors to processors for communication locality, computation/communication scheduling) from the programmer.

There have been broad projects that targeted optimization of all aspects of stream computation including the CPU organization (to include wide parallelism), bandwidth in the register/memory-hierarchy, and interconnection network [11], [5]. This work has a limited focus, which is to optimize the communication of streaming applications on existing mass-market multi-/many-cores, rather than on specialized streaming hardware. The key challenge in supporting stream communication on cache-coherent, shared-memory multi-/many-cores is that the communication patterns for coherence traffic (irregular, bursty, latency-sensitive) and streaming traffic (large, sustained transfers between known source/destination pairs, bandwidth-sensitive) are significantly different. Consequently, the interconnection networks designed for coherence traffic (irregular, bursty, short, fixed-size packets) are not well-suited for the communication requirements of streaming applications.

The above observation leads us to a dilemma – at one end of the design spectrum, we have specialized hardware which optimizes performance at the cost of the economies of scale of mass-market multicores, and at the other end of the spectrum we have mass-market cache-coherent, shared memory, multicores which offer poor support for streaming communication. The key goal of this paper is to develop efficient support for streaming communication with minimal, incremental changes to coherence networks of mass-market many-cores.

To motivate our design, we begin by describing the two key inefficiencies of coherence networks in supporting streaming communication. The first inefficiency is the mismatch between "pull-based" communication supported by coherence networks and the "push-based" communication used by streaming applications. To understand this mismatch, consider the least disruptive way to support streaming on coherence networks of shared memory multicores. In such a design, streaming would be implemented using the underlying shared memory (i.e., by implementing streams as software-based FIFOs) and no hardware changes would be needed. Coherent shared memory supports pull based communication where the producing actor deposits new values in its local caches, which are then "pulled" by coherence mechanisms by consuming actors. In contrast, stream communication is inherently push-based wherein producing actors actively push data to consuming actors. Supporting push-based communication using pull-based mechanisms wastes bandwidth because pull-based communication requires a request and a response, rather than a simple data packet. In addition, because pull-based communication requires producer-consumer synchronization, there may be other packets to be exchanged as well.

In contrast, we could implement a push-based network by allowing the processor to directly place data in the local input queues of the router. The packet would then be injected into the network and carried to the destination node. At the destination node, the recipient actor would similarly pull data directly out of the exit-queues of its local router. Thus, push based networks result in reduced communication and improved bandwidth for streaming applications. Networks that support a push-based interface can be implemented with minimal changes to existing coherence networks. However, implementing push-based communication alone is insufficient because of the second mismatch between the requirements of streaming communication and coherence networks.

Streaming applications require large, sustained data transfers and are bandwidth sensitive, unlike coherent shared memory workloads which have bursty, irregular, lateny-sensitive communication with short messages. Coherence networks are very efficient within their domain of operation: they use packet-switching to maximize link utilization for short coherence packets; they use minimal routing to reduce latency. We make two observations on this coherence-vs.-streaming mismatch. First, because of the bandwidth-sensitivity of streaming applications, minimizing network link contention for each stream, even at the cost of increasing hop count, is a desirable goal. Second, because streaming applications communicate large amounts of data over extended periods of time between known pairs of actors, it is perfectly acceptable to incur some overheads to setup contention-free routing paths. The overheads will be amortized over the duration of the streaming application (or the duration between actor migration, if such migration is permitted). Previous orchestration mechanisms have typically assumed simple (coherence-network like) routing algorithms as a constraint. For example, the StreamIT compiler[1] assumes dimension-ordered routing (DOR) when optimizing the layout of actors on computational nodes. Such routing algorithms suffer from unnecessary link-contention which results in degraded application throughput. While minor routing changes like using minimal adaptive routing instead of DOR can help link-contention for coherence traffic, adaptive routing is less useful for streaming because of the in-order delivery semantics of streams. Adaptive routing may deliver packets out-of-order, thus requiring additional complexity for re-assembly. More importantly, even if packet re-assembly were free of cost, we show that while marginally better than DOR for some benchmarks, adaptive routing falls significantly short of the throughput achieved by our design described below.

The first contribution of this paper is that we identify routing of streaming data among communicating actors as a novel degree of freedom in optimizing stream applications (unlike previous techniques which assumed a fixed routing algorithm). Based on the above observation, we propose a family of flexible routing algorithms – disjoint path routing algorithms – that customize routing for each streaming application as a key component of stream program orchestration. Disjoint-path routing algorithms aim to minimize link contention and thus improve the throughput of streaming applications.

Our second contribution is the design of a specific instance of disjoint-path routing called *tentacle routing*. Tentacle routing operates in a distributed fashion to discover and setup contention-free paths between communicating actors. Tentacle routing is a best-effort mechanism that is not guaranteed to find such disjoint paths even if they exist. In spite of that, it is attractive because (a) it works well in practice, successfully finding disjoint paths for all the benchmarks we consider and (b) when tentacle routing fails, we may always fall back on the underlying routing algorithm. Tentacle routing leverages circuit-switching as the switching mechanism which can be efficiently supported with incremental changes on

1. Specifically, the compiler backend for the Raw machine.

a packet switched router, as demonstrated by Jerger *et.al.* [10], [9]

Finally, we evaluate tentacle routing over a suite of streaming applications. Tentacle routing results in significant throughput improvements (on average, 55%–84% for applications with high link contention and 28%–41% for all applications) over two different system sizes. Most importantly, tentacle routing achieves near-ideal throughput (within 6% for 4x4 and within 11% for 6x6 networks). The combination of high throughput and simple implementation makes tentacle routing a very attractive design point.

In summary, the major contributions of this paper are:

- We identify and describe a design space of disjoint-path routing algorithms that directly aim to minimize link-contention among communication streams of streaming applications.
- We design one concrete instance of disjoint path routing called tentacle routing that can leverage a pre-existing router design that requires minimal changes to a packet-switched router.
- Tentacle routing improves application throughput by 28% and 41% on average in 4x4 and 6x6 systems, respectively. It achieves within 6% and 11% of ideal performance on 4x4 and 6x6 systems, respectively.

The rest of the paper is organized as follows. Section 2 describes our disjoint-path routing techniques. Section 3 describes our experimental methodology. Section 4 presents the results. Finally, Section 5 concludes this paper.

2. Disjoint Path Routing

In this section, we describe our technique for disjoint-path routing for stream programming models. Though our techniques are broadly applicable to various streaming programming models and network topologies, we limit discussion in this paper to the StreamIT programming language [16] and to two-dimensional mesh networks. Two-dimensional meshes will be the likely connection-fabric as we move to the manycore era. First, we offer a brief background of orchestration for the StreamIT programming model (Section 2.1). Next, we describe the space of disjoint path routing techniques, in general, and examine the pros and cons of several points in the design space (Section 2.2). Finally, we describe our design – *tentacle routing* – which is one specific instance of disjoint path routing Section 2.3.

2.1. Background and Related Work

In StreamIT, applications may be interpreted as a graph, with the computation expressed as "actors" or "filters" (vertices S, J, $C1$, $C2$, $C3$ and $C4$ in the graph on the left of Figure 1(a)) that communicate among one another over streams (edges in the graph). The programmer may overdecompose applications and express them using an arbitrary number of actors irrespective of the number of processors. Because the actors may be imbalanced in terms of workload and because the actors may exceed the number of available nodes, the compiler automatically fuses/fisses actors to achieve a transformed, load-balanced, stream computation graph in which (a) the number of actors are comparable in terms of workload and (b) the number of actors does not exceed the number of processors.

Once the load-balancing is complete, actors in the stream computation graph are mapped to processors in the layout stage as shown in Figure 1(a). The StreamIT compiler [7] attempts to optimize layout of actors in the fabric by assuming that the underlying routing function imposes a communication cost and optimizing the overall communication cost via simulated annealing [12]. Specifically, StreamIT's Raw processor back-end uses a cost-function that penalizes distance (number of hops) and switch interference (number of streams that pass through a switch). Assuming X-first, dimension-ordered routing, we can observe that there is significant link contention on the "North" incoming link at node J in Figure 1(b). It is this link contention that our technique targets. One may think that the link contention is an artifact of dimension ordered routing and that adaptive routing can eliminate/reduce link contention. However, we will show later that adaptive routing is insufficient to eliminate link contention. Instead, our technique focuses on routing stream data on disjoint paths – paths that do not share any links – as shown in Figure 1(c). Note, the paths may be longer than minimal paths (e.g., the path between $C2$ and J, in Figure 1(c)) because of our insight that streaming applications are bandwidth sensitive (i.e., it is more important to avoid link contention) and not latency sensitive (i.e., it is okay to increase the number of hops).

2.2. Disjoint-Path Routing Design Space

The disjoint path routing problem may be stated in two different ways. The first variant, which is the most general statement of the problem, treats layout and disjoint-path routing as a single integrated problem. Given some stream graph (in which actors have been

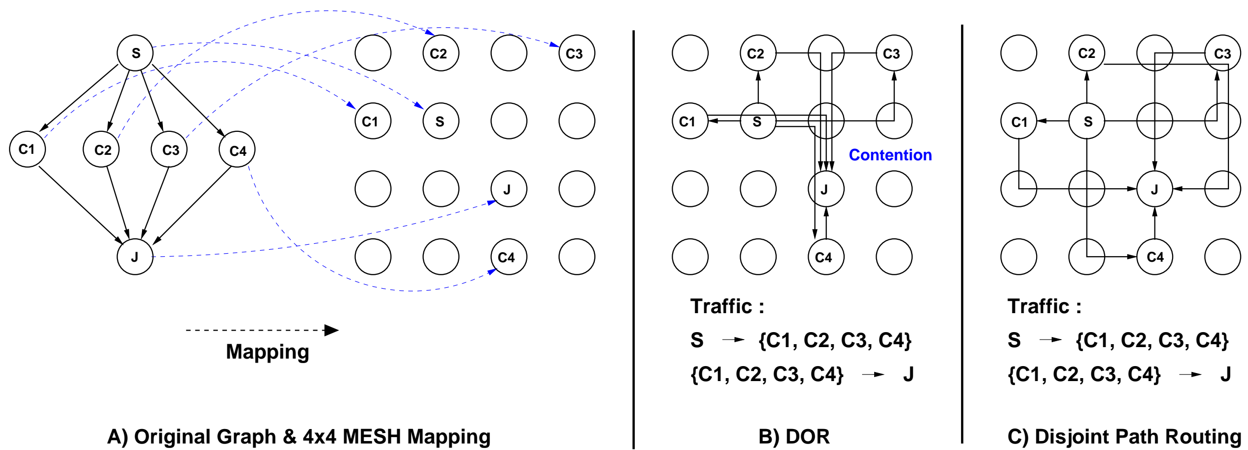


Figure 1. Maximizing Throughput via Disjoint Path Routing

load balanced) and some network topology, what is the optimal way to map actors in the graph to nodes in the network topology and edges in the graph to network paths to eliminate network contention? This is the same as the problem of finding a node- and edge-disjoint embedding of the stream graph in the network topology graph. Because such embedding of arbitrary graphs is a hard problem, we reformulate a simpler variant which can still capture the benefits of disjoint path routing. The second variant stipulates that there exists a mapping of actors to nodes (possibly from the layout obtained by simulated annealing assuming a simple routing algorithm). As a result of this simplification, disjoint-path routing is reduced to computing a set of paths from each source to its corresponding destination such that no two stream paths share a network link. One potential drawback of the above simplification is that it may result in an infeasible layout (i.e., a layout for which there is no feasible disjoint-path routing). However, such a scenario may be handled by retrying the routing problem after modifying the layout. While this simple approach does not upperbound the number of attempts needed, we found that it was practical approach as it was adequate for all the benchmarks we considered. Within this variant, we examine other design choices in terms of the following two questions.

Who is responsible for discovering the edge-disjoint paths? Given that the expected run-time of the application is significantly longer than the routing time, disjoint path route discovery can be done in software at one centralized node (at the compiler or runtime system). The route discovery can use sophisticated routing algorithms such as those used in VLSI netlist routing. For example, disjoint path routing on a mesh network may be viewed as equivalent to global routing on a rectilinear grid. Each stream corresponds to a two-terminal *net* and the entire set of streams corresponds to the netlist. Using the above formulation, powerful techniques like *concurrent routing* may be brought to bear on the problem [8], [3]. On the other hand, such heavy-weight computation may introduce too much overhead especially if the runtime system allows actor migration. An alternative would be to use simpler routing algorithms such as net-by-net routing.

While the above algorithmic options were being considered at a central node, it is also possible to let each node attempt to discover edge-disjoint routes in a distributed fashion. Such a design inherently requires the net-by-net (stream-by-stream, in our context) routing approach in which routing of nets occurs independently without global, algorithmic co-ordination. Further, such net-by-net routing may be sequential or parallel. One challenge when implementing net-by-net routing in a parallel fashion is deadlock- and livelock-avoidance. If we assume link-by-link reservation for each path (because each path will consist of multiple links and because multiple links cannot be reserved atomically), there can be deadlocks. If we allow pre-emption and/or release of network links upon deadlock (“rip-and-route”, in VLSI routing terminology), there may be livelock. Finally, some layouts may not have any feasible disjoint path routing at all. Parallel net-by-net routing mechanisms must also have a consensus mechanism where all nodes agree that there is no feasible disjoint path routing. Alternately, net-by-net routing may also be implemented in a sequential manner. Sequential net-by-net routing (without pre-emption) is deadlock- and livelock free since there are no other contenders reserving

links concurrently. However, routing for one stream may fail because all necessary links are held by previously routed streams. In such cases, some mechanism to detect failure when there are no feasible paths is needed.

In all cases of failure (either because disjoint path is truly infeasible or because of algorithmic artifacts in net-by-net routing) we may fall back on the underlying network's routing algorithm because it is acceptable to have degraded performance rather than to not perform at all.

What are the switching mechanisms for Disjoint Path Routing? Once such a set of disjoint paths are discovered, we must deploy appropriate switching mechanisms to enable packet transmission along those paths. Possible switching mechanisms include circuit switching or source-routed packet-based switching. both of which closely match stream requirements (in-order delivery along fixed network paths, with bandwidth-reservation in case of circuit-switching). Further, recent research has shown that circuit switching can be supported via minor modifications to the canonical packet-switched router [10], [9]. Most importantly, because the modified router allows packet switching and circuit switching to co-exist, it is an ideal architecture for multicore on-chip networks, where multiple programming models may be used. Source-routed packet switching can also be supported with very few incremental changes to a canonical packet-switched router.

2.3. Tentacle Routing

While the above discussion focused on the broad principles of disjoint-path routing and the need to minimize link-contention, in this section, we describe a concrete instance of disjoint path routing called tentacle routing.

In Tentacle routing, route discovery is distributed. Consequently, we use the stream-by-stream routing rather than a globally co-ordinated concurrent routing strategy. Further, route discovery is sequential and not concurrent. Each node attempts to discover routes for its own outgoing streams in strict sequence by using a greedy algorithm, with backtracking. Route discover across nodes is also sequential. The operation of tentacle routing algorithm is illustrated in Figure 2. Each node sends out control flits along the minimal paths reserving links as they progress. If links are available, the entire stream path is established (e.g., the paths between the $< S1, D1 >$, and $< S1, D2 >$ pairs in Figure 2(a)). However, if an attempt to make forward progress fails because of unavailable links (i.e., links reserved by other streams) the control flits backtrack to the previous node and are rerouted along other links. For example, the attempt to establish a path from $S2$ to $D3$ initially reserves the "North" link to node $S1$. But upon discovery that there are no free outgoing links from $S1$, it backtracks to node $S2$ (Figure 2(b)) from where it tries the "East" port (Figure 2(c)). As routing proceeds, the non-minimal routes (detours) may also be chosen if the greedy (minimal) choice is unavailable.

Handling deadlocks, livelocks and routing failure. Because tentacle routing is sequential and non-preemptive, it is guaranteed to be deadlock- and livelock-free. Because of backtracking, tentacle routing effectively does an exhaustive depth-first search, which makes it easy to detect failure when tentacle routing fails. Tentacle routing is a best-effort routing technique because it does not guarantee that a disjoint-path routing will be found even if one exists because of our stream-by-stream routing approach. The failure to discover feasible routes can occur because tentacle routing incrementally adds paths for communicating pairs. Since the routes discovered for earlier pairs are not relinquished until all paths have been established, the probability of success may depend on the order of consideration. While such failures are possible, in all our benchmarks, tentacle routing discovered disjoint paths without fail. Because of the difficulty of generating meaningful "perturbed" layouts (StreamIT compiler's output layouts are very consistent, and random layouts are not meaningful), we do not present any empirical sensitivity study of failures. We leave a theoretical analysis of the possibility of failures for future work. As mentioned earlier, upon such failures, it is possible to consider other options including retrying routing after changing (a) the order of stream consideration and (b) the layout (i.e., the actor-to-node assignment). As a final safety net, we may also declare that no feasible path exists and fall back on the underlying routing algorithm as stated before. Note, to prevent deadlocks between the stream paths and regular packet-switched packets, we assume that *all* streams fall back to the underlying routing algorithm if *any* stream fails to be routed.

We adopt, with two modifications, a hybrid router architecture proposed recently by Jerger *et.al.* [10], [9] that can support both packet-switched traffic and circuit-switched traffic. The focus of the work by Jerger *et.al.* is on optimizing coherence performance via circuit switching where pairwise coherence communication is predictable. Our focus is entirely different; we wish to use the circuit-based switching mechanism to deliver packets along the disjoint paths that Tentacle routing discovers. Tentacle routing requires two key changes in the router behavior described by Jerger *et.al.* [10], [9].

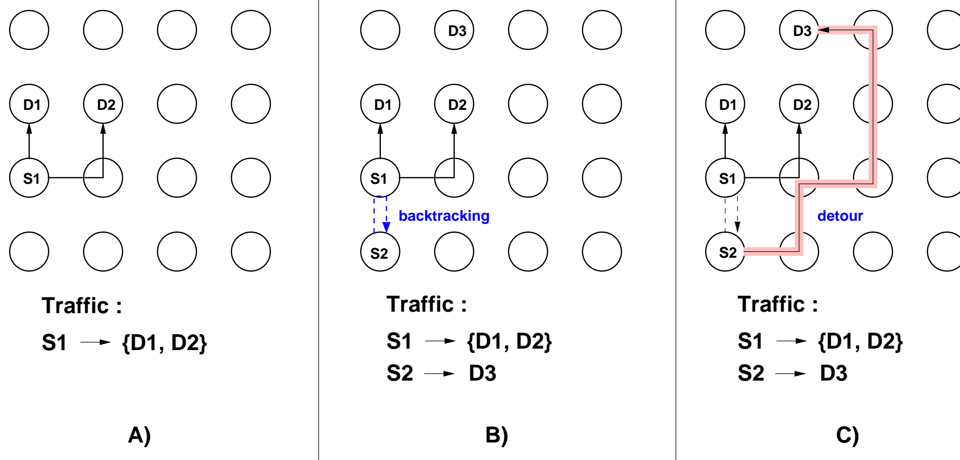


Figure 2. Tentacle Routing: An Example

First, the circuits we setup are "hard" resource reservations, unlike the "soft" reservations used in their design. In their design, established circuits may be dissolved when there are more circuits than the switch can handle. In the absence of this change, our backtracking-based route discovery mechanism does not work (because links are never unavailable). Second, the setup network must be modified to support Tentacle's greedy routing with backtracking. Their design leaves routing unchanged, and only alters switching. The hardware support for routing-with-backtracking has been discussed elsewhere [6] in the context of fault-tolerance. Note, our novelty claims are limited to tentacle's disjoint-path routing mechanism; we do not claim novelty on the hardware support for circuit-switching/backtracking.

3. Methodology

Streaming Applications and Network Load Generation. We use the StreamIT compiler[16], [17] and the set of stream applications distributed by the StreamIT project (listed in the first column of Table 2). We use the StreamIT compiler's built-in functions to create load-balanced sets of actors that are then laid-out on our networks (4x4 and 6x6 mesh networks). Because the built-in layout functionality of the StreamIT compiler in customized for the RAW machine's static network, we introduce a minor modification in the layout computation to accurately model our network configuration. We modify the cost function used in simulated annealing to remove the synchronization cost since our routers can handle multiple incoming flits without stalling. To generate the network workload corresponding to streaming applications, we use a front-end driver that (a) reads inputs from the input stream queues (waiting if necessary for inputs to be available), (b) loops for a duration that's proportional to the work at the actor, and (c) emits output data to its output stream queue. We vary the proportionality constant (*computation scaling factor*) to alter the network load. (This corresponds to varying the computational power of the processor.) We assume that memory bandwidth is not a bottleneck. Memory may be accessed in two different ways. First, there may be memory accesses by the computation in actors/filters that go off-chip. However, we model all computation (including memory stalls) as a single delay. Second, the input stream data and output stream data will have to go off-chip to be written to memory. We do not model the bandwidth of such memory accesses in our simulation since there are known techniques to offer large memory bandwidths (e.g., stream buffers) for bulk, contiguous accesses.

Router Configurations. We simulate two system sizes (4x4 and 6x6) and six sets of router/switching configurations for each size. The six routing/switching configurations are as follows: two push-based packet-switching configurations (one with DOR, and another with minimal adaptive routing algorithm), two pull-based packet-switching configurations (one with DOR, and another with minimal adaptive routing algorithm), one tentacle routing configuration and finally an "ideal" configuration. The various network configurations (with additional network parameters) are listed in Table 1.

Pull-based streaming mimics the network behavior of streaming using coherent shared memory. In such systems, the communication of any unit of data has

to occur via two network packets which carry the coherence request and the cache-block response, respectively. Note, the above approximation is a conservative simplification because it ignores other coherence network traffic such as (a) the invalidates produced by the producers' writes and (b) the request-response packets for a synchronization variable which must be used for accurate producer/consumer synchronization. We consider two variants of pull-based configuration; one with DOR and another with minimal adaptive routing. Because adaptive routing can result in out-of-order packet delivery, it introduces the additional burden for stream re-assembly. However, we conservatively assume stream re-assembly for free.

Push-based streaming assumes some interconnection network interface wherein the processor can directly inject contents into the network's local queues (e.g., stream registers or memory mapped stream queues). With such interfaces, actors can directly push data into a stream which is then delivered in the local queue at the destination node across the network. As with pull-based streaming, we examine both DOR and minimal adaptive routing.

For tentacle routing, we assume the router architecture described in the previous section with a push-based processor/network interface. The choice of circuit-switching vs. source-routed packet-switching is not expected to make a big difference in performance since the bulk of the improvement occurs because of link-contention elimination. Though Tentacle routing is not guaranteed to find a disjoint-path network, tentacle routing discovered disjoint paths in all our benchmarks. (In general, we may fall back on the underlying packet routing mechanism if tentacle routing fails.)

Finally, we also assume an "ideal" configuration in which all actors can communicate with "downstream" actors with a latency of one cycle. We model the bandwidth of links and the total incoming/outgoing bandwidth of each node. The ideal configuration effectively represents a customized network whose topology matches the stream graph. Special-purpose streaming network architectures cannot outperform the ideal configuration.

Simulator. We use a modified version of PoPnet[14] network simulator. PoPnet models a four-stage pipelined router for packet-switching network. Once the circuits for disjoint paths are setup, the packets go through only two stages (switch traversal and link traversal). The network was simulated for 100,000 cycles which was seen to be adequate for the network to achieve steady state. And before collecting data, there is 10,000 cycle warm-up time. The channels are full-duplex bidirectional links.

Network Configuration	
Topology	2D Mesh
Network Size	4x4, 6x6
Routing Algorithm	DOR, Adaptive
Switching Method	Packet(Push, Pull), Circuit
VCs/PC	2, 4

Table 1. Network Configurations

Benchmark	Link contention	
	4x4 Network	6x6 Network
Audiobeam	High	Low
Beamformer	High	High
Bitonic Sort	Low	High
Channelvocoder	High	High
DES	Low	Low
FilterBank	High	High
FMradio	High	High
TDE	Low	Low
Vocoder	Low	Low

Table 2. Benchmarks and their classification according to link contention

In terms of buffer resources, circuit-switching network requires a single virtual channel, because circuit-switching network does not suffer from HOL (Head-Of-Line) blocking. In contrast, adaptive routing algorithm for packet-switching network requires at least two virtual channels. Further, to avoid HOL blocking, we evaluate packet-switching network with four virtual channels. This effectively means that our comparisons of tentacle routing with packet-switched routers are very conservative since the packet switched routers use twice the buffer resources as tentacle routing's circuit-switching network.

4. Results

The four primary conclusions of our results are:

- Five of the nine applications have significant network contention which results in increased opportunity for tentacle routing. (Details in Section 4.1.)
- Adaptive routing helps to reduce link contention marginally, but is unable to eliminate it. Further, as expected, all flavors of push-based communication are more efficient than pull-based communication resulting in improved throughput for push-based configurations. However, push-based communication alone (without disjoint path routing) suffers from a 37% throughput degradation compared to

tentacle routing. (Details in Section 4.2.)

- On a 4x4 (6x6) system, tentacle routing achieves an average (geometric mean) of 55% (84%) throughput improvement for applications with high network contention which corresponds to 28% (41%) throughput improvement overall. On a 4x4 (6x6) system, tentacle routing achieves within 6% (11%) of an ideal communication architecture. (Details in Section 4.3.)
- Tentacle routing has very low overhead, even under extremely pessimistic assumptions (e.g., 0.015% overhead assuming reconfiguration on every OS tick). (Details in Section 4.4.)

The remainder of this section elaborates on these key results.

4.1. Measuring Opportunity: Link Contention of Streaming Applications

Recall that if stream paths are disjoint under dimension-ordered routing, there is no opportunity for disjoint-path routing to improve streaming performance. In this section, we classify the benchmarks based on the link contention assuming dimension-ordered routing and demonstrate that tentacle routing does achieve higher performance on applications with high contention.

Given the layout generated by the StreamIT compiler, we assume dimension-ordered routing, and use the number of streams that map to a link as a metric of contention. Figures 3 (a) thru 3(d) illustrate the link contention for four selected applications out of the nine applications we consider; two high-contention applications(`filterbank` and `FMradio`) and two low-contention applications (`vocoder` and `bitonic-sort`) on one network configuration (4x4). In each graph, we plot the links on the X-axis (the order of appearance of the 48 physical links is not important) and the number of streams that use that link on the Y-axis. Among the high-contention applications (`filterbank` and `FMradio`) we observe several links with contention. For example, link 33 in Figure 3(a) and links 28 and 33 in Figure 3(b) have five streams that contribute to link load. In contrast, the link loads for the low contention applications `vocoder` and `bitonic-sort`, never exceed one[2], which implies that paths obtained by dimension-ordered routing are naturally disjoint. Consequently, we cannot expect tentacle routing to improve communication performance any further.

Due to lack of space, we omit detailed link-load distribution results for the other five applications. Instead, we present summary results in Table 2 with a broad classification of high-contention applications and low-contention applications, as measured on 4x4 and 6x6 networks. Note that there are minor differences in application classification in the two network configurations (4x4 and 6x6) because the stream graph and its placement are dependent on network size. The next section examines how the opportunity measured by link contention translates to improved application throughput with tentacle routing.

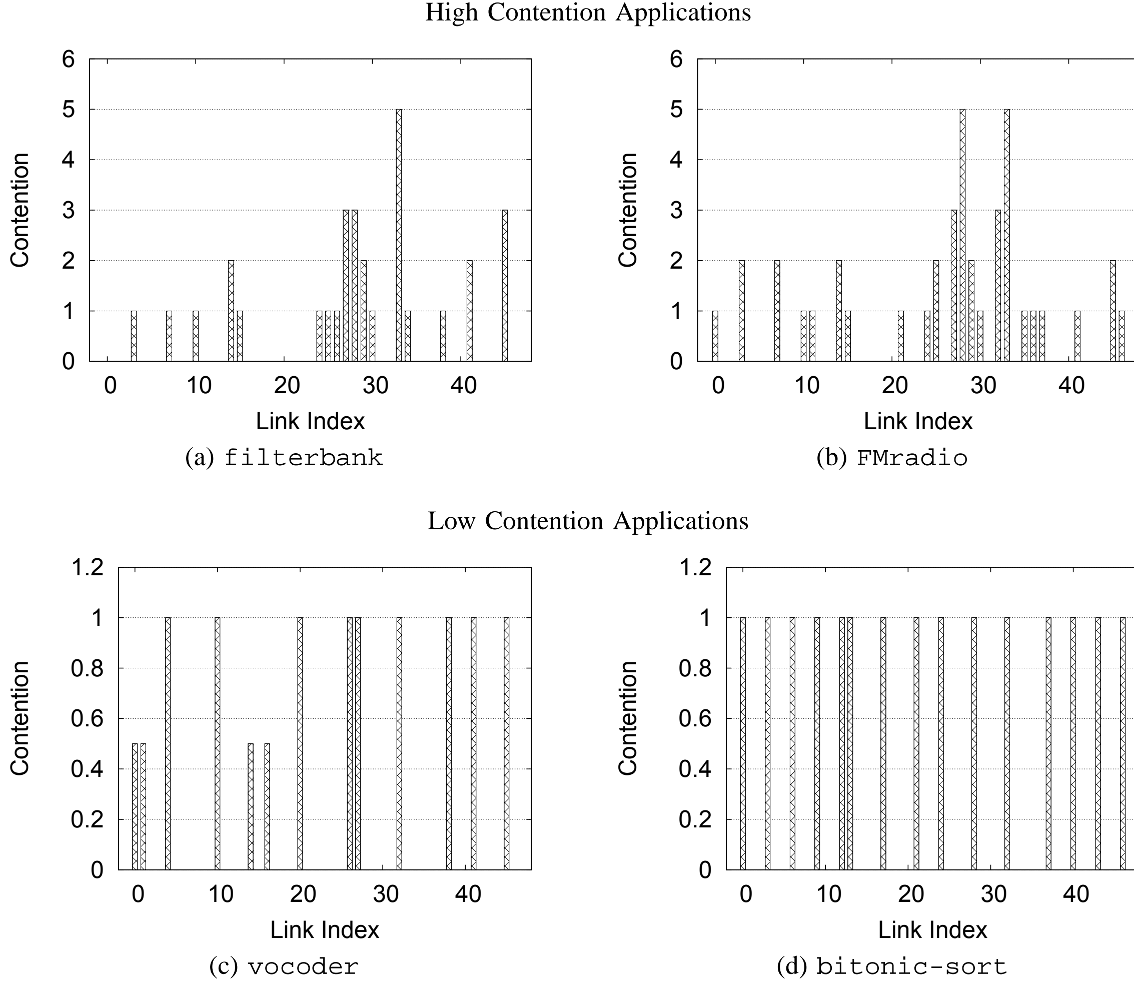
4.2. Performance Analysis of Network Configurations

The four graphs in Figure 4 illustrate the performance of the same set of four applications. Each graph contains six curves for each of the six routing/switching configurations. The X-axis plots the computation scaling factor (used to scale the computational workload of actors, as described in Section 3). For uniformity, we normalize the computational scaling factor to 1 at the lowest scaling factor where the application is still computation-bound. As can be seen in Figure 4, the application throughput is largely independent of network configuration when the computational scaling factor is 1 or greater. When computation is scaled to smaller numbers, computation ceases to be a bottleneck and we enter the communication-bound region of operation where the network configuration affects overall application throughput (shown in absolute units on the Y-axis). Note that lower values on the X-axis correspond to higher network loads. Further, because the normalization of the computation scaling factor is application dependent, the computational scaling factor cannot be compared across applications; it is meaningful only within a single application. Similarly, the absolute values of throughput (on the Y-axis) are not comparable across applications.

In the communication-bound region of operation, as load increases, the throughput increases though it eventually saturates at different levels for different routing/switching configurations. One may observe a clear difference between the pull-based networks and push-based networks (with push-based networks being better). This is not surprising since pull-based networks effectively impose a bandwidth penalty on streaming communication.

For the low-contention workloads (Figure 4(c) and Figure 4(d)), there is no difference in throughput among

2. Because we count fractional link loads when a single stream is demultiplexed over several links, it is possible for link load to be less than one.



(a) `filterbank` (b) `FMradio`

(c) `vocoder` (d) `bitonic-sort`

Figure 3. Link Load Distribution

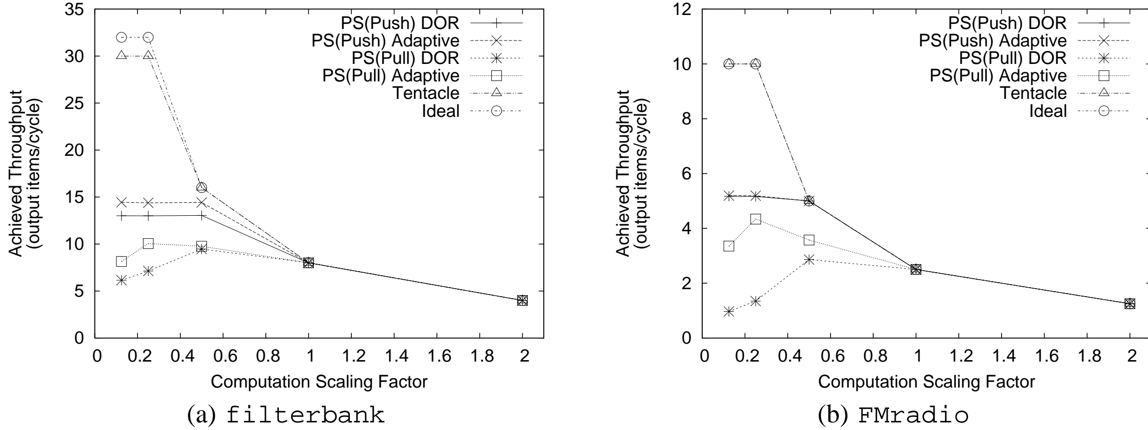
tentacle routing, the two other push-based mechanisms, and the ideal configuration, even in the communication-bound region. This is as expected because of lack of opportunity.

In contrast, the channel load of `filterbank` and `FMradio` show significant link contention(Figure 4(a) and Figure 4(b)). Not surprisingly, the throughput improvement for these benchmarks is also significant. Further, we observe that adaptive routing results in a small throughput improvement for `filterbank`. However, the improvement is significantly less than the improvement achieved by tentacle routing. Finally, tentacle routing achieves near-ideal performance in both high-contention applications.

4.3. Performance Improvement

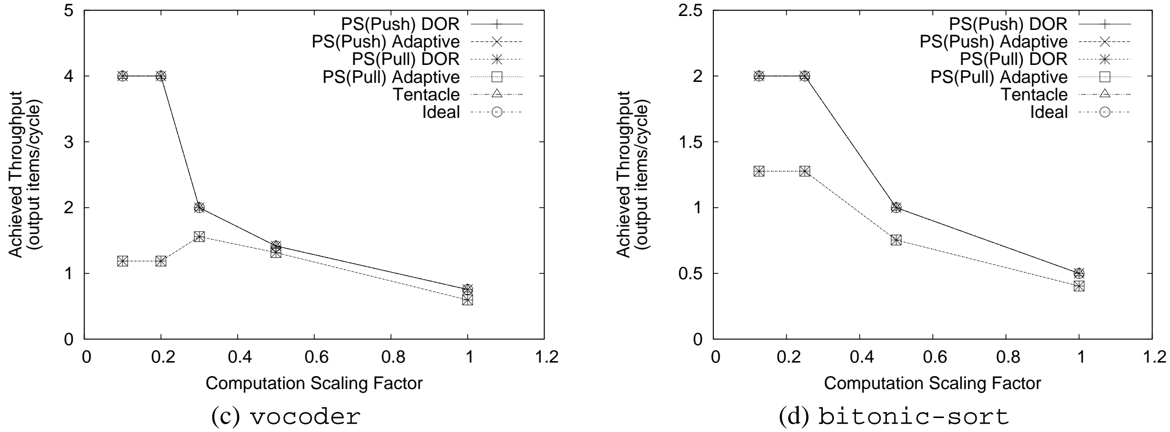
Figure 5(a) and Figure 5(b) plot the throughput improvements for all nine streaming applications on the 4x4 and 6x6 system, respectively. Each figure includes six bars for each streaming application (shown on the X-axis). Each bar corresponds to one of the routing/switching mechanisms. The height of the bars (Y-axis) shows the maximum throughput achieved by that particular configuration (i.e., the saturation throughput in the communication-bound region) normalized to that of the configuration with push-based packet-switching and adaptive routing (the best prior scheme). In addition, we include two additional sets of bars; one shows the geometric mean of normalized throughput across all high-contention applications and another shows the geometric mean across all applications. The benchmarks are listed in the order of increasing link-contention.

High Contention Applications



(a) `filterbank` (b) `FMradio`

Low Contention Applications



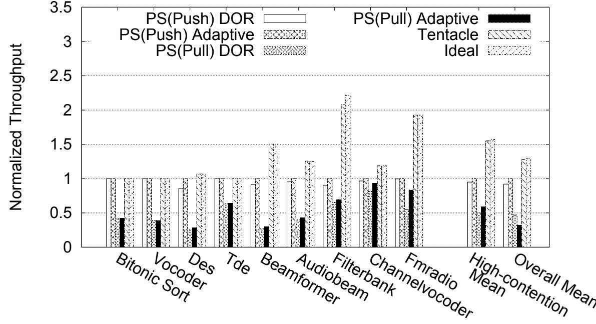
(c) `vocoder` (d) `bitonic-sort`

Figure 4. Peak Throughput for routing/switching configurations

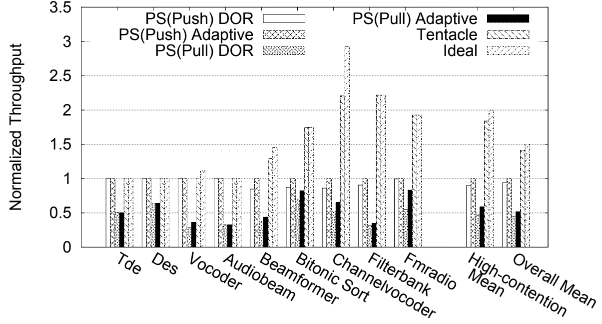
Not surprisingly, we observe that the best improvements are for the applications with the most link contention. When considering only high-contention applications, tentacle routing results in 55% throughput improvement for the 4x4 system and 84% for the 6x6 system. When averaged over all applications, the mean throughput improvement is 28% and 41% for the 4x4 and 6x6 systems respectively. Also, tentacle routing achieves near-ideal throughput (within 6% and 11%) in the two network sizes. The gap between tentacle routing and ideal is because of the single-hop latency that the ideal configuration enjoys.

4.4. Tentacle Routing Setup Overhead

Typically, the streaming routes are set up once at the beginning of application, following which the same stream computation is applied continuously for huge amounts of data. For example, the FMradio application may typically run for several minutes/hours. As such, tentacle routing overheads may be amortized over the entire run of the application. Table 3 quantifies the overhead (in cycles) of setting up the various disjoint-path routes for 4x4 and 6x6 networks. For each benchmark, the minimum (second column), maximum (third column) and average (fourth column) single path setup times are shown. The total overhead shown in fifth column includes the overhead of setting up all paths, one after the other. The absolute overhead varies between 605 (741) cycles and 2009 (3109) cycles on a 4x4 (6x6) network which is neglible as a fraction of execution time because streaming applications run on large streams of data for minutes/hours whereas the setup overhead for tentacle routing is no greater than 1.5 microseconds assuming a 2GHz clock.



(a) 4x4 System



(b) 6x6 System

Figure 5. Throughput Improvement

In rare situations, when the run-time forces actor migration or when application changes to add new filters (e.g., addition o noise-removal filters in DSP algorithms when channel noise increases), paths may have to be re-formed. Though our experiments did not re-configure routes at run-time, we observe that the low overheads of path setup can easily accommodate reconfiguration on every OS tick (say 10ms) since the overhead is about 1.5 microsecs (i.e., 0.015% overhead). Thus, we conclude that the circuit establishment overhead of tentacle routing is negligible.

Benchmark	One Path Establishment Time			Total Overhead
	min	max	average	
4x4 network				
Audiobeam	5	13	7.29	605
Beamformer	5	21	7.97	1613
Bitonic Sort	5	5	5	1505
Channelvocoder	5	31	10.29	1125
DES	5	17	6.41	1709
FilterBank	5	59	11.61	826
FMradio	5	33	12.2	2009
TDE	5	9	6.07	1509
Vocoder	5	9	5.33	1205
Average				1345
6x6 network				
Audiobeam	5	41	13.46	741
Beamformer	5	29	9.25	2729
Bitonic Sort	5	21	6.52	2917
Channelvocoder	5	65	11.17	1625
DES	5	17	6.41	1709
FilterBank	5	41	14.18	2241
FMradio	5	33	12.2	2009
TDE	5	9	5.26	3109
Vocoder	5	18	6.88	3105
Average				2242

Table 3. Disjoint Path Routing/Establishment Overhead (cycles)

5. Conclusions

Streaming is emerging as an important programming model for multicores. Streaming provides an elegant way to express task decomposition and inter-task communication, while hiding laborious orchestration details such as load balancing, assignment (of stream computation to nodes) and computation/communication scheduling from the programmer. Previous work on orchestrating streaming programs treated a given routing algorithm as an immutable constraint and attempted to maximize application throughput within that constraint. This paper enables novel optimizations based on the observation that routing need not be viewed as an unchangeable constraint; instead flexible routing can be viewed as a new degree of freedom in orchestrating streaming applications.

Our design is driven by two observations. First, streaming application performance is bandwidth-sensitive. From this observation, we conclude that assigning stream communication to disjoint network paths minimizes network link-contention, and thus maximizes both the bandwidth available for streaming communication and application throughput. Second, streaming communication involves large, systematic data transfers between known communicating pairs of nodes over extended periods of time. From this observation, we conclude that incurring some overhead to compute and setup the disjoint network paths is justified since (a) the costs are amortized over large periods of time and (b) the benefits of disjoint path routing are significant. Based on the above two design decisions, we

develop one instance of disjoint-path routing called tentacle routing – a backtracking, best-effort, circuit-routing technique. On a 4x4 (6x6) system, tentacle routing results in 55% (84%) and 28% (41%) mean throughput improvement for high-network-contention streaming applications, and for all streaming applications, respectively.

Acknowledgments

The authors would like to thank anonymous reviewers for their feedback. This work is supported in part by National Science Foundation (Grant no. CCF-0541385).

References

[1] The Cell Project at IBM Research, http://www.research.ibm.com/cell.

[2] NVIDIA CUDA Zone, http://www.nvidia.com.

[3] L. Behjat and A. Chiang. Fast integer linear programming based models for vlsi global routing. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 6238–6243 Vol. 6, May 2005.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[5] W. J. Dally et. al. Merrimac: Supercomputing with streams. In *SC'03*, November 2003.

[6] Patrick T. Gaughan and Sudhakar Yalamanchili. A family of fault-tolerant routing protocols for direct multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):482–497, 1995.

[7] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[8] Jiang Hu and Sachin S. Sapatnekar. A survey on multi-net global routing for integrated circuits. *Integration, the VLSI Journal*, 31:1–49, 2001.

[9] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Circuit-switched coherence. *Networks-on-Chip, International Symposium on*, 0:193–202, 2008.

[10] Natalie Enright Jerger, Mikko Lipasti, and Li-Shiuan Peh. Circuit-switched coherence. *IEEE Computer Architecture Letters*, 6(1):5–8, 2007.

[11] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[13] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2008. ACM.

[14] L. Shang, L. S. Peh, and N. K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proceedings of the 9th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 79–90, Feb 2003.

[15] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[16] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, April 2002.

[17] William Thies, Michal Karczmarek, Michael Gordon, David Z. Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. A common machine language for grid-based architectures. In *ACM SIGARCH Computer Architecture News*, June 2002.

[18] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, 2008.