Signaling Networks
Order-Graph Sampler

Changelog

0.0	7/3/10	Initial draft.
0.1	7/10/10	Added partial result accumulation (MCU).
0.2	7/12/10	Updated R/E questions section.

Chris W. Fletcher
Narges B. Asadi
7/3/2010

Introduction

The following is a memorandum detailing version 3 (V3) of the MCMC order-graph sampler design implemented on FPGAs. Major changes and optimizations across different versions are chronicled below and on the right side of this page.

	V1	V2	V3
System			
Order sampler	■	■	■
Graph sampler		■	■
SW←→ Support (RCBIOS)		■	■
[Latency hiding] LS/PS load time			■
Multi-chip support (PLiN)		■	■
[Latency hiding] cross-chip			■
Multi-restart support		■	■
Parallel tempering	■		■

	V1	V2	V3
SN			
Hardware LS/PS scheduler			■
Hardware local order scheduler			■
ST			
Multi-port support		■	■
Multi-lane support			■
[Latency hiding] cross-core			■

	V1	V2	V3
MCU			
Threading across restarts			■
Local order coalescing			■
Node caching		■	■
Order caching		■	■
Local order scheduler			■
Embedded GPP-based approach			■
Partial core/node result accumulation			■

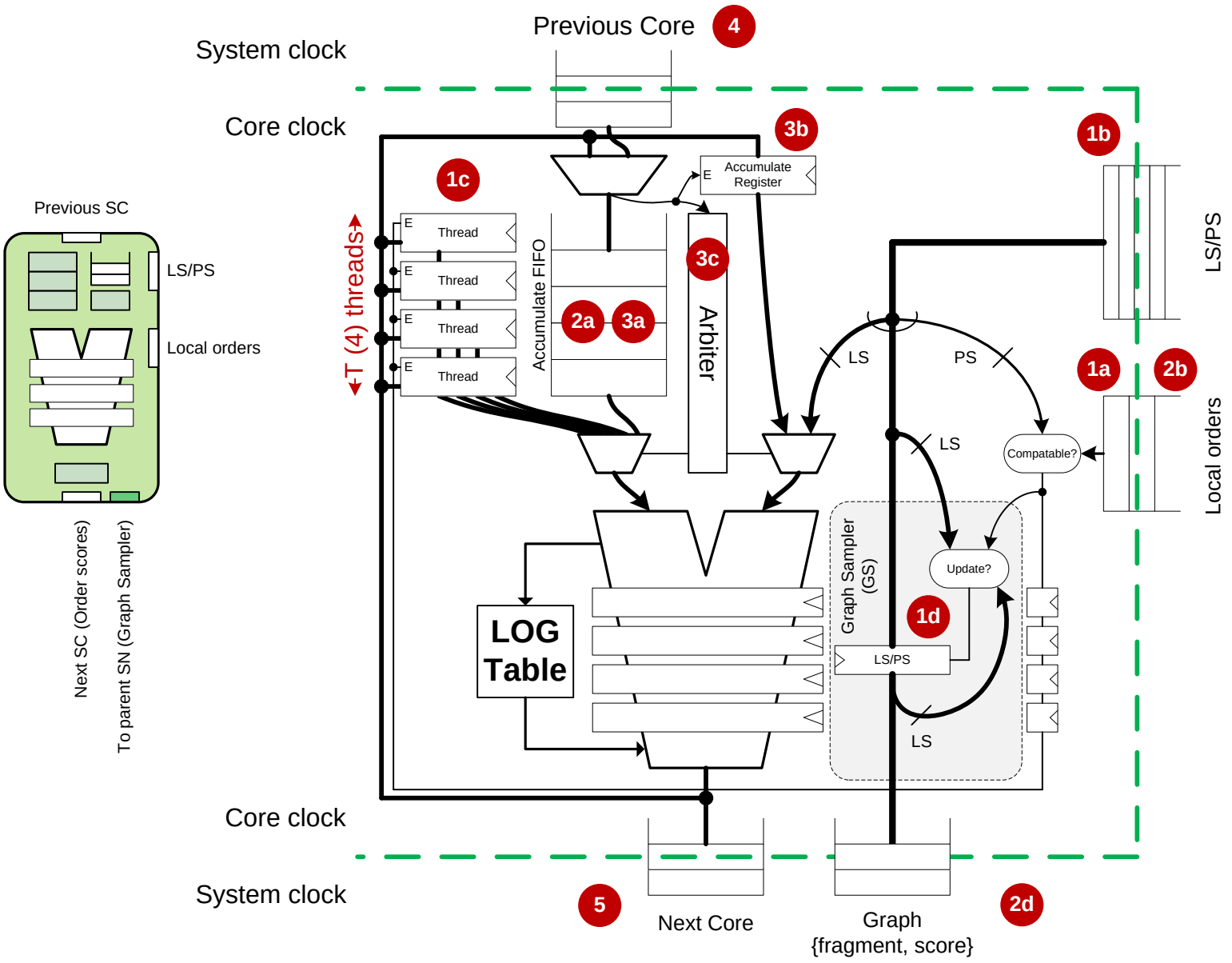
	V1	V2	V3
SC			
Hardware fine-grained multithreading		■	■
Optimized `log` tables		■	■
Alternate `log` table implementation			■
{Core, system} clock decoupling			■
[Latency hiding] cross-thread			■

	V1	V2	V3
MSU			
SN-decoupled scoring logic			■
DRAM-backed LS/PS paging		■	■
Parallel multi-SN DRAM loading			■

V1: FPGA'08 paper.
V2: ICS'10 paper.
V3: proposed here.

■ Not implemented
■ Proposed, not viable
■ Proposed, research Q
■ Implemented, viable, and/or well understood

Scoring Core (SC)



The Scoring Core's (SC's) responsibilities are to

(a) score a local order against a subset of the parent node's parent sets & local scores.

(b) accumulate the partial score with that of the previous node, and pass the new partial score onto the next core.

and (c): determine the compatible parent set whose local score is greatest, and return both that parent set and local score.

(a) and (b) constitute the Order Sampler (OS) while (c) implements the per-core Graph Sampler (GS).

This SC design enhances the V2 design by hiding the "result accumulation" step in the scoring process. Specifically, the V2 SC had a T^2 + T*C cycle result accumulation overhead (C = the number of SCs in an SN lane). This implementation has a T+1 cycle overhead in the steady state (when 2+ local orders need to be scored), and an absolute T^2 + T*C latency (when only one local order is being scored).

Operation

1a-d.) **[When a local order arrives and the Accumulate FIFO is empty]** When the core receives a local order (1a), it iterates over the LS/PS FIFO (1b) and accumulates the LS (for each compatible PS) across T fine-grained threads (1c). This takes approximately PPC cycles. The compatible PS whose LS is greatest is saved (1d), per the GS step.

2a-d.) **[When a local order has been fully processed]** When each of the T threads is dispatched for the last time, each is committed to the Accumulate FIFO (2a). As the T threads are being FIFO-committed, the next local order is set (2b) and the T threads, that have committed, restart. As the last thread is committed, the LS/PS saved in (1d) is sent to the GS results buffer (2d).

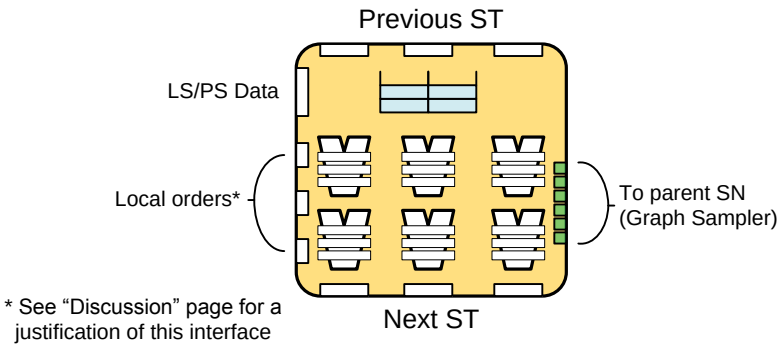
3a-c.) **[When a local order has arrived and the Accumulate FIFO is not empty]** As the next local order is scored, the Accumulate FIFO (3a) and Accumulate Register (3b) take control of the pipeline for one cycle every T cycles (3c). The T cycle delay is due to there being a serial dependency between accumulating results in the Accumulate Register.

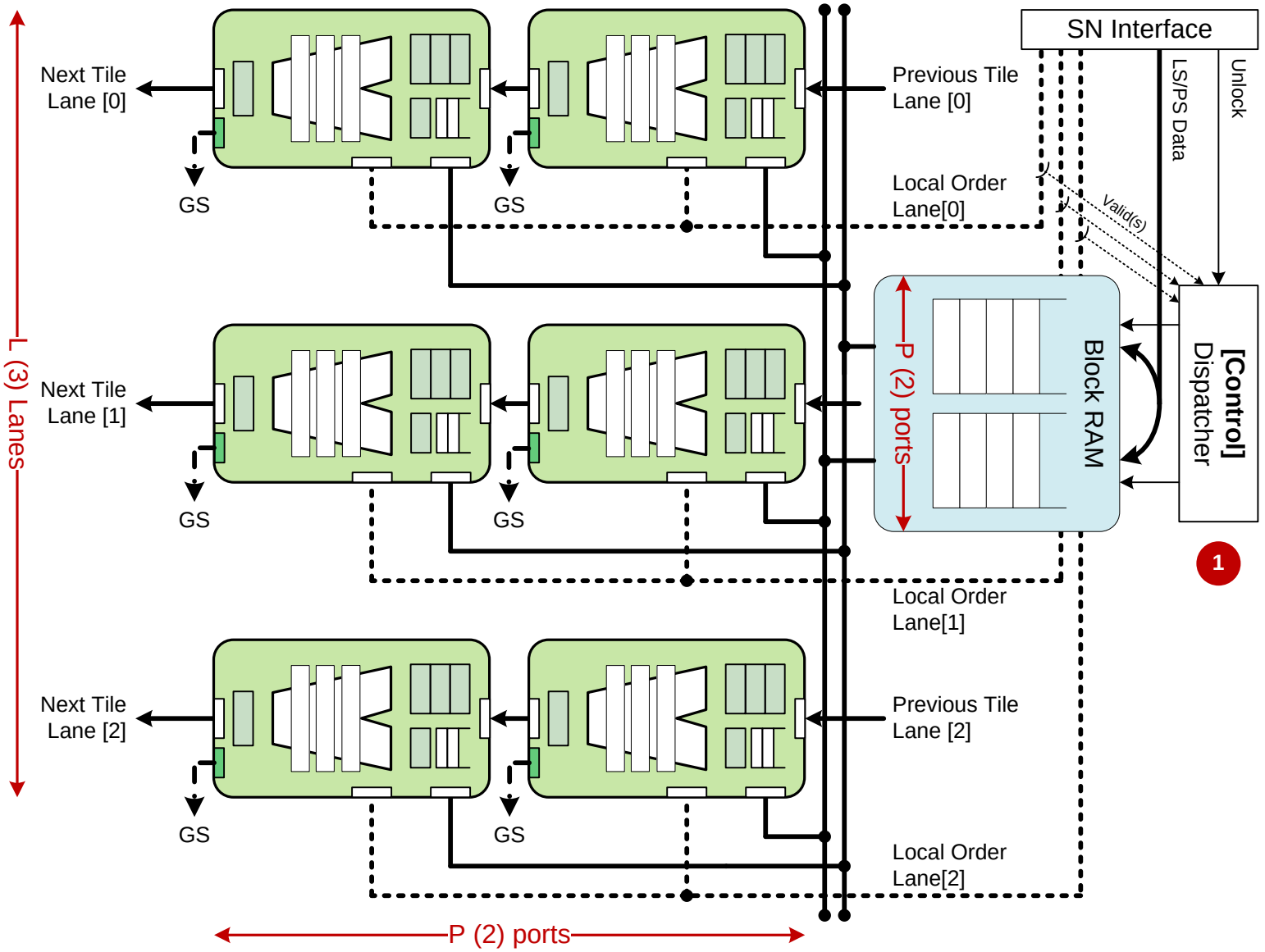
In between when Accumulate FIFO contents are injected into the pipeline, the current local order continues to be scored. Thus, the accumulation process adds T cycles to the scoring process (per local order) in the steady state.

4.) When a result from a previous core arrives, it is added to the Accumulate FIFO.

5.) As the last element in the Accumulate FIFO exits the pipeline, it is passed to the next core.

Scoring Tile (ST)





The Scoring Tile (ST) is a collection of SCs around a block RAM unit (BRU). Over each BRU, there is one SC per BRU port (typically 2 for Virtex-5/6 FPGAs), multiplied by the number of lanes supported by the parent SN. SCs along a lane share a local order and are chained together, implementing the result accumulation step as a linear reduction. SCs across lanes snoop off of the same LS/PS data that flows from the BRU.

The ST’s only control logic is the “Dispatcher” (1), which is responsible for:

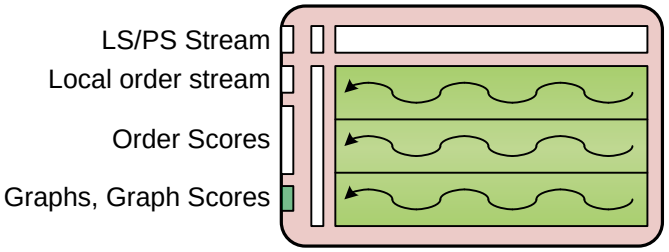
(a) Knowing when each lane has at least one local order.

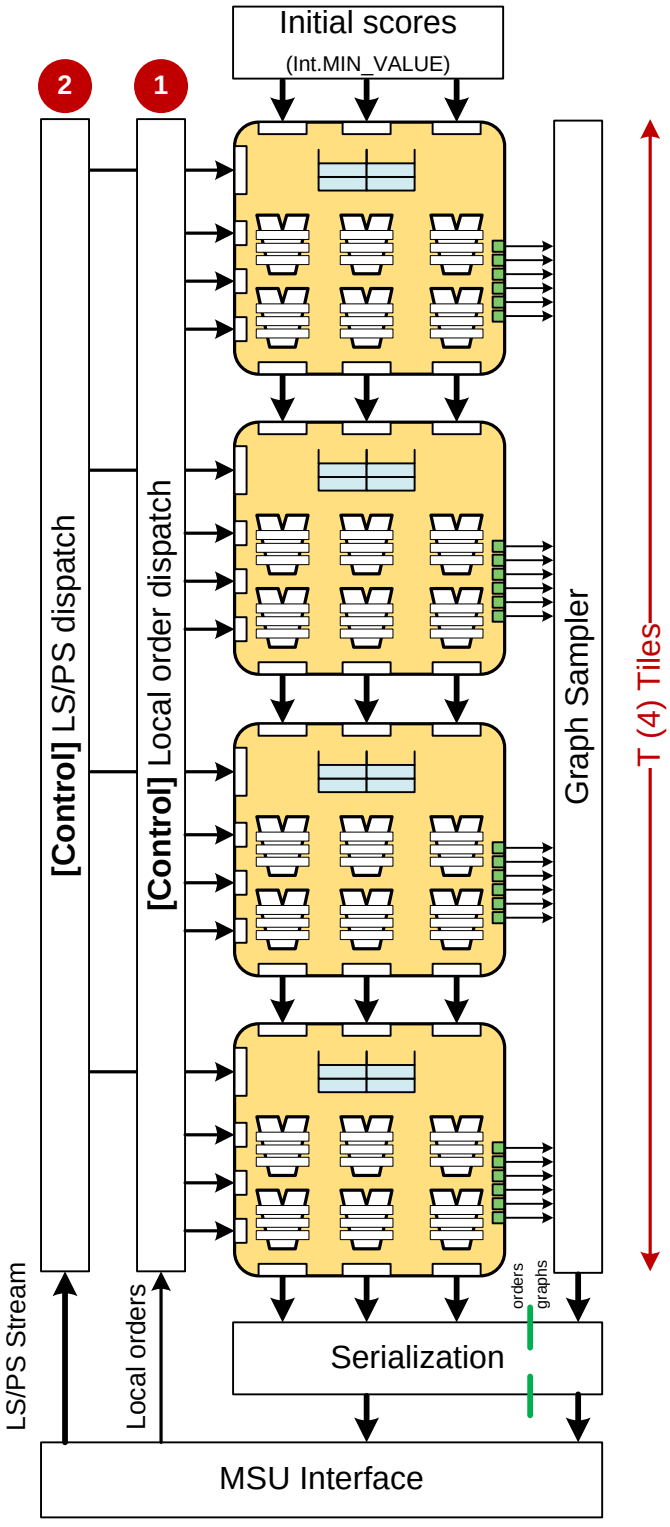
(b) Iterating through the BRU when (a) holds.

(c) Restarting when (b) finishes.

Implementation note: When there are no local orders across any lanes, the dispatcher should start scoring when local orders arrive and either (a) each lane is assigned a local order, or (b) the SN unlocks the tile. When the local order buffers are partially full, the ST should restart the BRU as needed until they are completely empty. The ST can schedule by statically knowing the PPC and keeping a counter for each lane’s local order count (or through watching the `empty` signals coming from each core’s local order FIFO).

Scoring Node (SN)





The Scoring Node (SN) is the collection of STs necessary to score a complete node. That is, the set of BRUs in the SN must accommodate the PPN requirement for the node that is being scored.

The SN is oblivious to the node that it is actually scoring. It receives a stream of local orders and a stream of LS/PS data from the Scoring Network (MSU). Its job is to:

(1) Evenly distribute (round robin stripmining) local orders across ST lanes. By implication, the outside system does not need to address a lane when sending a local order to get scored. **[Note: Lanes and ports within an ST are not decoupled; thus, the SN must either ensure that every lane has a local order or alert the ST (through an 'Unlock' signal) that one or more of the lanes will be unused for this set of local orders.]**

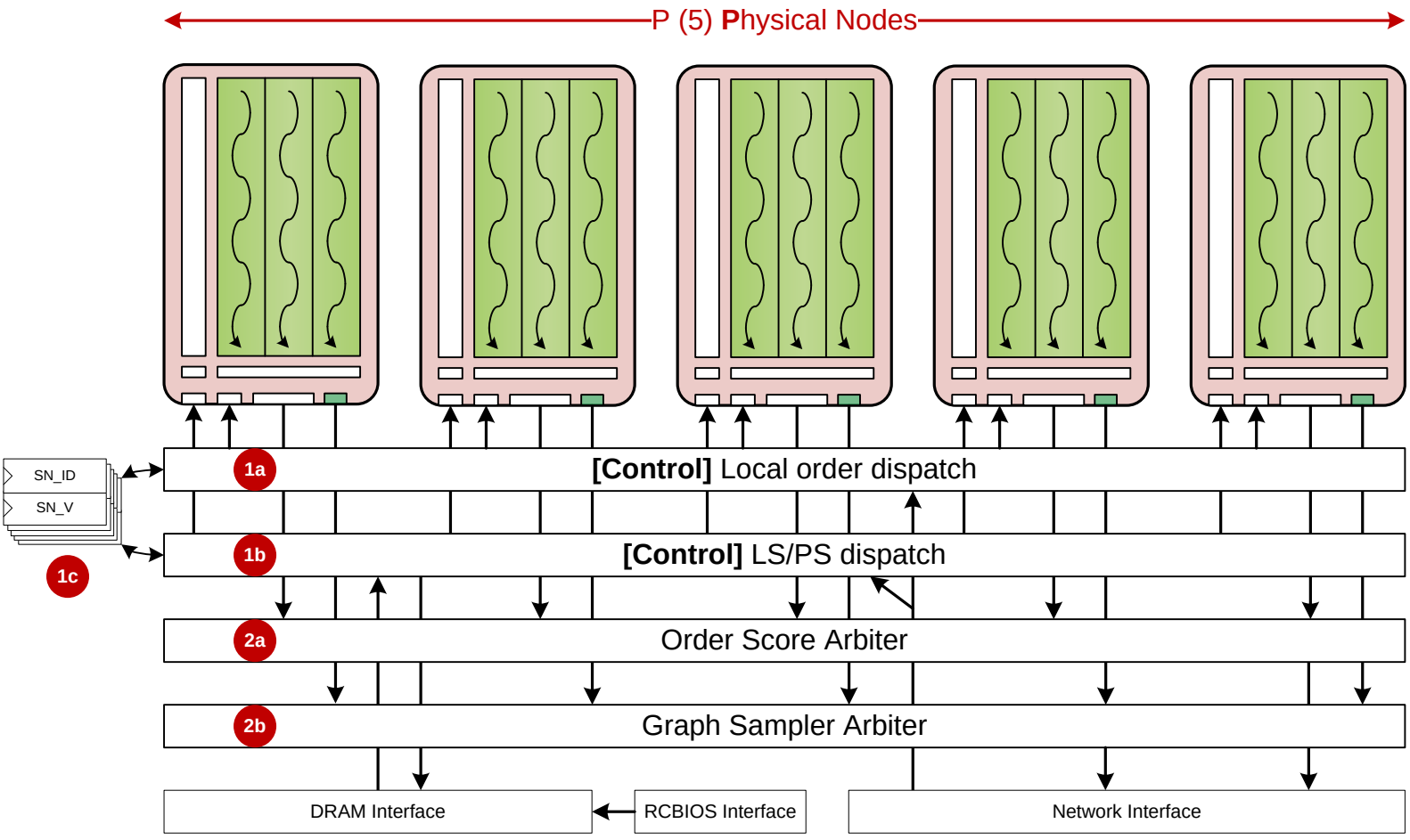
(2) Evenly distribute (round robin stripmining) LS/PS data across the ST BRUs. This allows the ST to runtime optimize for an arbitrary node, given the PPN for that node (which is provided at the start of a new LS/PS load). **[Note: The SN's performance will be bottlenecked by the ST with the most LS/PS.]**

Graph Sampler

The SN GS unit takes the LS/PS GS result from each core in each lane and keeps the LS/PS pair whose LS is greatest (this is the same as the SC GS operation) for each lane.

Implementation note: Two straightforward ways of implementing the GS reduction across cores is through a shift-register/comparator/MUX at each core or a high fan-in and pipelined MUX/comparator that takes as inputs the LS/PS results from each core. On a Virtex-5 FPGA, the high fan-in MUX implementation is ~ 2x less area, and is what is currently used and shown (interface-wise) in this diagram.

MCMC Scoring Network (MSU)



MCU → MSU

The Scoring Network (MSU) is a collection of SNs, scheduling, memory DMA, and routing logic. Each chip/FPGA has at most one MSU. At system initialization time, the LS/PS for every node in the system is loaded into the DRAM that is connected to each MSU. Thus, each SN in each MSU can score any node after that node is 'paged' in.

MSUs operate when they receive this message from an MCU:

SN Mask	R	NID	Off	Len	1+ Local orders ...

This packet issues local order commands to one or more SNs and [optionally] refills the SNs' BRUs with a new node's LS/PS data.

Fields:

SN Mask: An SN_count-width mask that indicates which SNs should be refilled / tasked to score the attached local orders.
R: "Refill" - the active high LS/PS data refill flag.
NID: Indicates which node's LS/PS should be "refilled" into the SNs indicated by the SN Mask.
Off: Indicates at what offset in the node's LS/PS the refilling should start from.
Len: Indicates how many LS/PS should be loaded.
Local orders: A string of 1+ local orders, which are to be streamed and scheduled across the SNs' scoring apparatuses.

It is the MSU's responsibility to schedule how local orders within the message are distributed across available SN resources (1a) and when the SN refill occurs (1b).

Implementation notes:

1.) Local orders should be distributed across SNs in group round robin style, where a group is as large as the number of lanes supported by each SN.
2.) When multiple SNs are to be refilled, **the LS/PS data that refills one can be used to refill the others at the same time**. Thus, the time it takes to refill any number of SNs with the same node's LS/PS is independent of the number of SNs being refilled.

Control maintains a register for each SN that indicates 1.) what node's LS/PS data is held at that SN, 2.) that the SN is loaded with valid data (SN_ID, SN_V, respectively ~ 1c). When an MSU receives the MCU local order message, it behaves based on the state of these registers:

R	SN_ID	SN_V	
0	Don't care	1	Schedule & score payload
1	Don't care	0	Refill, Schedule & score payload
1	Doesn't match 'Node ID'	1	Schedule & score payload, Refill

The "score, refill" case is similar to pre-fetching the next node.

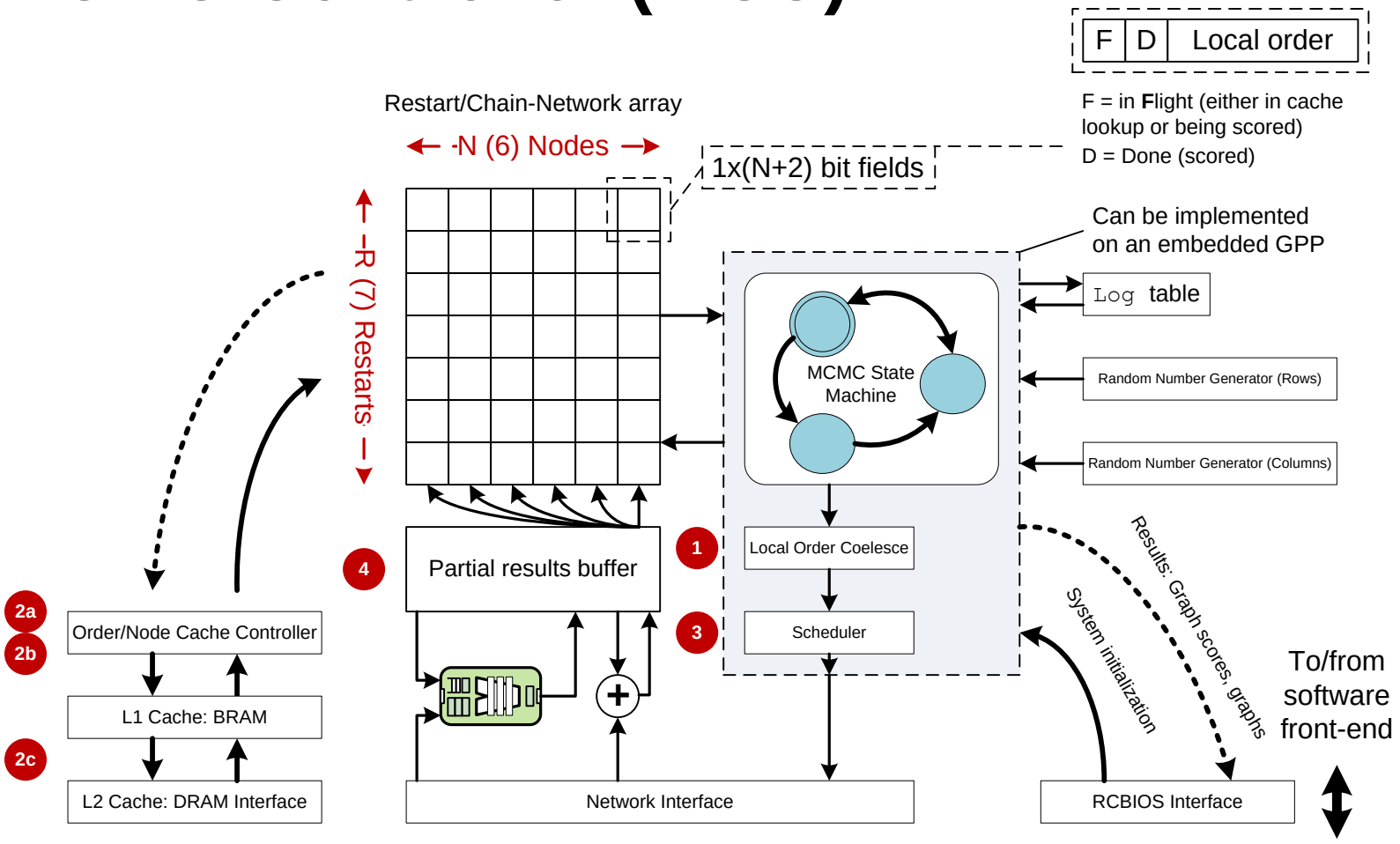
MSU → MCU

Once a local order is done being scored, its {order, graph} scores and graph fragment are sent back to the MCU, tagged with the original local order in a packet of the following structure:

Local order	{Order, Graph} Score, Graph fragment

(item ordering in payload is arbitrary). The MCU associates the local order tag with one or more restarts, and applies the results to those restarts.

MCMC Controller (MCU)



The MCU provides system control for one or more MSUs. At a high level, the MCU keeps track of different order/restart threads (each of which is made of N local orders). The MCU first scores as many of each restart's local orders as possible on its own, and then schedules the remainder to be scored across available MSUs. The MCU scores on its own through "coalescing" (1) and caching (2):

1.) **Local order coalescing** takes a vertical column out of the Restart-Network array and identifies duplicate local orders. Since these duplicates will result in the same {order, graph} scores and graph fragments, the duplicates are eliminated and only one is ever sent to the MSUs to be scored.

2.) Caching works because a given local order (for a certain node) will always yield the same {order, graph} node scores and graph fragments. [**Note:** Local order coalescing should spread cache hits around to duplicate local orders.] Caching key-values/latencies are as follows:

> **Order caching (2a):**
> {order}→{{order, graph} scores, graph}
> All lookups take ~ (DRAM latency) + R cycles.
> **Node caching (2b):**
> {nodeID, local order}→{node {order, graph} scores, graph fragment}.
> All lookups take ~ (DRAM latency) + N^2 cycles.

Caching lookups start as soon as a new iteration begins, and independently from local order coalescing. The order cache is used first because of its potential to reduce the local order count dramatically per lookup and because it takes few cycles to complete (relative to the node cache lookups). The caches can be implemented in DRAM or through an L1/L2 BRAM/DRAM inclusive system* (when the MCU does not share chip real estate with an MSU, which can be the case with multi-chip systems). Local order coalescing proceeds as and after cache lookups occur so that the system doesn't have to wait for coalescing to complete (more necessary if coalescing isn't directly implemented in hardware).

When the MCU has shrunk the set of local orders as much as possible, it schedules the remaining local orders across the MSUs.

3.) Scheduling can be implemented directly in hardware or through a GPP scheduling algorithm (3). In hardware, complexity grows quickly past a simple round-robin scheme. In software, the time it takes to schedule will have an impact on performance. Regardless of the implementation, the scheduler can take advantage of the following [relevant] information in the system:

3i.) As long as 1+ local order for a given node needs to be scored, that node **must** be scheduled onto an MSU.
3ii.) An SN can score L local orders in parallel (L is the ST lane count).
3iii.) The MCU can calculate how long it takes to score a local order for an arbitrary node through knowing the PPC for that node.
3iv.) The MCU can calculate how long it takes to refill an SN with a node's LS/PS through knowing that node's PPN.
3v.) More than one SN in the system can score a given node (or a fraction of a given node). **By extension, stripmining nodes across FPGAs/chips in the system is a good strategy for keeping all of the DRAMs active at a given time.**
3vi.) SNs that share an MSU can be loaded with the same node's LS/PS in parallel.

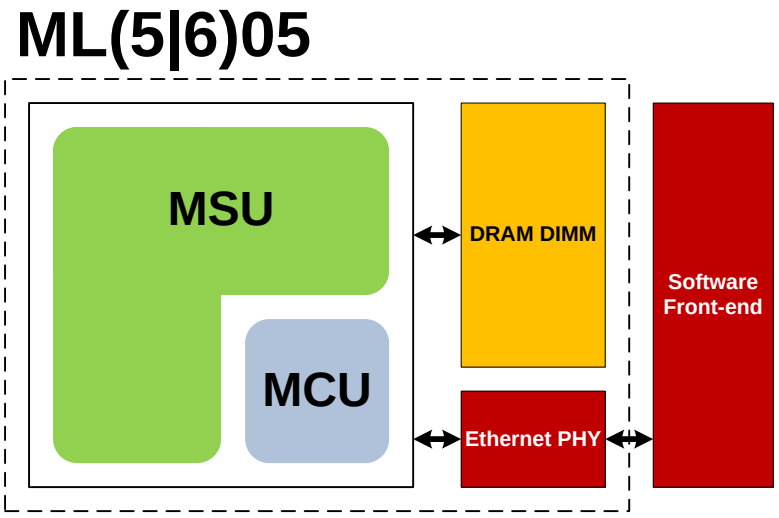
When deciding whether or not to allocate more than one SN to a particular node, there is the following set of tradeoffs:

a.) Allocating a new SN to a node incurs the LS/PS refill delay. Thus, it might be faster to just schedule the unscored local orders on an SN that already has the node of interest paged in.
b.) Allocating multiple SNs to a single node decreases the other nodes' abilities to be scored.

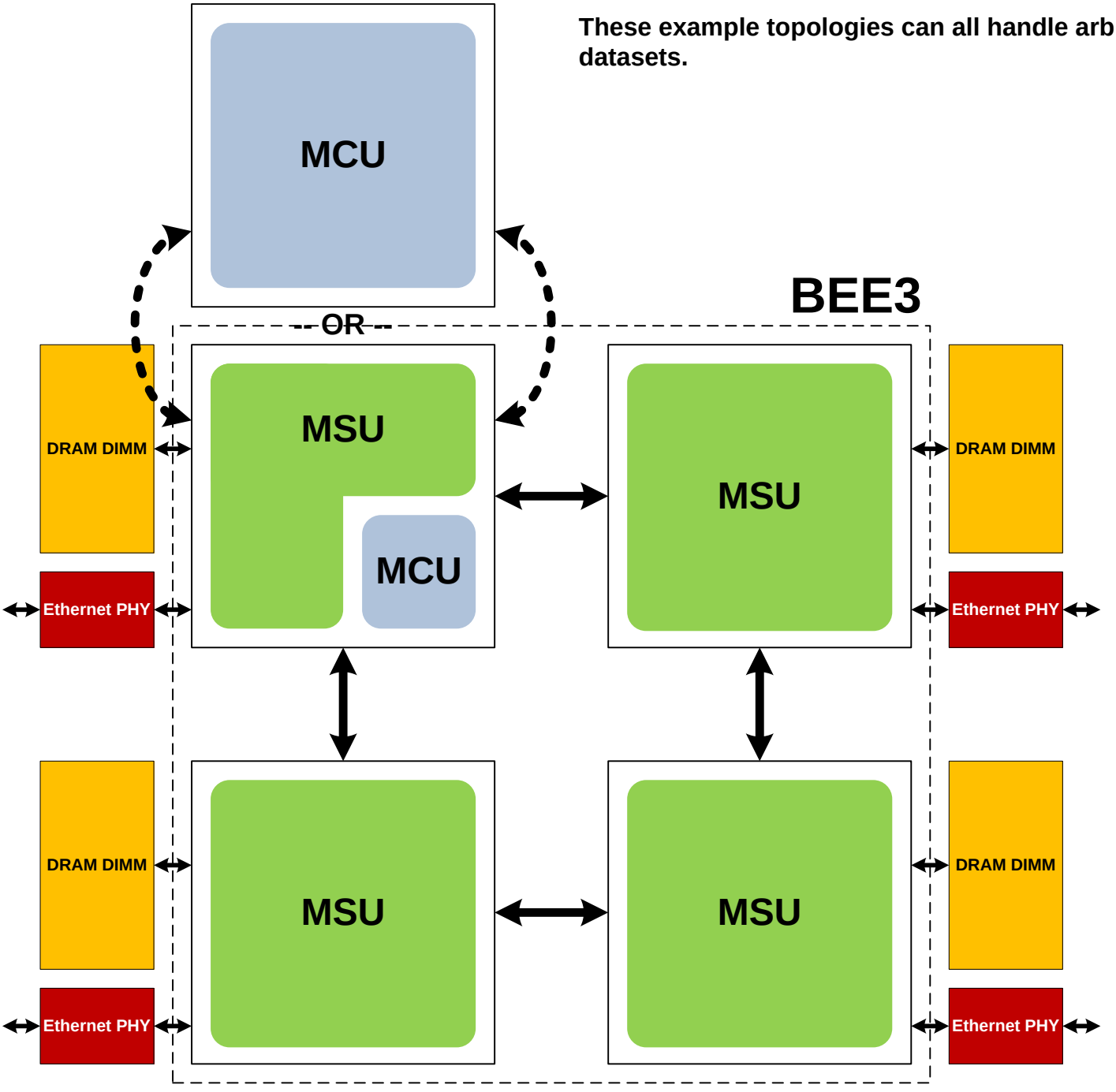
4.) For situations when a node is split across multiple SNs, the MCU accumulates per-SN results through a buffer that centralizes partially accumulated results. Since results are serialized back to the MCU, the accumulators built as part of (4) need not be heavily replicated to maintain performance.

Implementation note: As scored results return to the MCU from MSUs, they may arrived in interleaved FLIT order / be subject to other all:1 network problems. Reassembling interleaved streams can be done with a conceptually simple O(N) hardware solution (used in the ICS design). This circuit is mentioned because it can incur large hardware cost for an R restart system, and helps motivate separate MCU/MSU bitfiles.

System-level View



These example topologies can all handle arbitrary datasets.



Research Questions

1.) **Delta from FPGA Computer to ASIC.** There is nothing especially FPGA-specific about the MSU (or the MCU for that matter). What is the design cost in porting both the MSU and MCU to a custom ASIC? The goal would be to -as easily as possible- realize the increased clock frequency and lower power/area consumptions inherent in ASICs.

2.) **Standard forms in computer architecture.** The MSU follows standard computer architecture forms (specifically a SIMT machine featuring SIMD SNs). These types of machines (SIMD, MIMD, SIMT, VT, etc) often rely on micro architectural-level constructions (which expose thread, task, algorithm, etc level parallelism) in order to achieve better performance – as opposed to using lower-level application-specific circuits. Therefore, the same basic design blocks (FFs, arithmetic units, buffers) are prevalent in all of these types of machines. Thus, can we claim that if the delta to ASIC for the MSU/MCU design is small, that the delta will also be small for other designs that follow the standard forms in computer architecture?

If so, given the thrust to build FPGA computers out of architectural (MIMD, vector-thread) templates (cite: MARC), can we make the general claim that high performance FPGA computers can be quickly ported to ASIC* for better performance/power/etc?

*: FPGA to ASIC because the set of basic design elements available on ASIC is a super-set of what you would find on an FPGA.

3.) **System scalability.** Despite our ability to handle networks up to size 32, biologists will be providing data sets for networks up to size 100 (or more) in the future. This presents a hard scalability problem (32 node systems have 36457 parents per node, while 100 node systems have 3926176 -or- 75449320 -or- 1195978576 parent sets per node, depending on network indegree, which is likely to increase as network size increases). From an algorithmic perspective, there are cross-over points where (a) the architecture becomes constrained by different factors (bandwidth, computation, etc) and (b) employing different techniques (such as parent set filtering), etc start to become worthwhile. What are these cross-over points? For (a), how would an 'ideal' architecture change at each cross-over point? For (b), would accelerated implementations of the different techniques (such as the pre-processing step or parent set filtering techniques) move the cross-over points enough to be worthwhile?

3.) **Kernel interface & accumulator machines.** Although the MSU design is specific to the BN application, the math within the kernel can be swapped out to implement loop nest accumulator machines. Given a variety of memory access patterns, which might be relevant in accumulation-heavy applications, does the infrastructure around the MSU kernel need to be substantially changed?

3.) Partially developed ideas: loop nests as "seas" of function calls, loop nest analysis for deciding when to best reorder loops, … Note: the brunt of the partially developed ideas & thinking behind these are in Chris' paper notes...

Engineering Questions

1.) **Local order coalescing.** How often are there duplicate local orders for the same node, across all restarts? Does this effect become more pronounced when the different restarts start to converge? Or is the solution space 'peaky' to the extent that convergence says nothing of how similar one order/restart is to another?

2.) **Local order scheduler.** How much scheduling logic is necessary in order to make the scheduler effective? In practice, do the caches have balanced hit rates across different nodes, such that scheduling logic that maps more than one SN to a node is unnecessary? That is, can we get away with simple round-robin scheduling to good effect?

3.) **Embedded GPP approach.** How effective is an embedded GPP as opposed to full-custom scheduler and local order coalescing logic (this is based on 1 & 2, above)? How can we hide instruction latency when we need to (at the start of each iteration) and how much software complexity can we get away with when we are in the steady state (and only need to keep the SNs busy with occasional local order assignments)?

4.) **Alternate log table implementations.** Each SC implements log(1+exp(…)) using static 0-cycle-latency lookup tables. This is a major departure (and possibly advantage) from the GPP/GPU implementation, which goes to great length to avoid that function when it can because of the functions' compute times on those platforms. Right now, the FPGA can implement the table in between 125-150 LUT6. Given that this block is replicated per SC, can it be optimized using cordic functions or series approximations to greater effect?

5.) {**Core, system} clock decoupling.** In general, routing the V2 design higher than 200 Mhz has been difficult because of the heterogeneous accumulations needed across the design. When all of the SC logic is decoupled from the system by clock (which is acceptable because of the SC cycle latency in between needing new local orders), how good are the tools at PARing the SCs? How high can we get the SC clock in practice?

Discussion

The following are additional design observations, decisions and tradeoffs not discussed already.

1.) **Node caching.** The ICS design couldn't cache nodes well because a single node cache miss would render any other node cache hits ineffective, per iteration. V3 doesn't have this problem because of parallelization across restarts, but it will still be bottlenecked by the node that has the fewest cache hits, across all restarts. V3 compensates for this by being able to assign that node's LS/PS to more than one SN in parallel. So, the farther skewed the local order count is towards one node, the faster the system can score that node.

2.) **GPP-based MCU.** The MCU local order coalescing unit and scheduler may incur a large hardware cost in terms of both resources and design complexity if directly implemented in hardware. Using a GPP here would reduce both cost and complexity, yet decrease performance. The argument towards using a GPP is that it is only cycle-critical while SNs in the system are initially unscheduled. Once each SN is scoring some number of local orders, it will be hundreds of cycles before they will be able to process another set of local orders. Thus, making the GPP effective is a matter of quickly scheduling the first set of local orders, and then [possibly] taking longer to schedule the rest (in order to derive a more optimal schedule). Observation (3i) on the MCU page tells us that this is doable – since each node that has at least one local order needing to be scored **must** be scheduled on an SN at some point, scheduling the initial set of nodes can be done with minimal to no decision logic. Once those nodes are being scored, the scheduler can then take the time it needs to schedule the remainder.

3.) **Threading restarts across an SC.** There are two known techniques for hardware threading an SC – (a) the way presented already (where a single local order is broken up into more chunks and a cross-thread accumulation happens at the end, and (b) setting each thread to a different restart. (b) is attractive because it eliminates the need to cross-thread accumulate, which was a bottleneck in the ICS'10 paper. With the cross-thread accumulation scheme introduced in this document, however, where cross-thread accumulation is hidden by the next local order, (a) is superior. With (a), the SC can start scoring with less local orders present. Furthermore, if not enough local orders are available, (b) has pipeline bubbles. (a) is always guaranteed to fill all threads. Lastly, (b) requires T times more GS logic, whereas (a) requires a single GS pipeline per ST lane.

4.) **ST interface.** The ST interface exposes a separate port for each lane's local order input. Thus, the parent SN must schedule which lane gets which local orders. This might seem inelegant – **why shouldn't the ST decide this on its own and hide its lane count from the parent SN?** *If the SN sends a stream of local orders to different STs and the STs schedule the local orders on inconsistent lanes, the system will break.* The only way to provide the one-port local order interface is to have the ST scheduler be deterministic (such as through round robin). This is a reasonable alternative interface. That said,

(a) since the schedule is deterministic, having it made at the SN level makes it a one time area cost. Pushing it to the ST level makes the area cost grow with ST_count.

(b) the ST's other ports (lane outputs and GS results) also depend on the lane count. This means that even with hiding the lane count from the local order interface, certain parts of the SN will still have adapt as the lane count changes.

5.) **BRAM/DRAM vs. DRAM caches.** The BRAM/DRAM cache system is used to reduce the initial bulk cache access latencies (allowing the system to reach steady state faster) or to parallelize cache accesses. This latency hiding can also be achieved by speculatively swapping nodes in every order and performing the next iteration's cache lookups before the previous iteration is complete. Neither of these implementations have been tested. The BRAM/DRAM scheme is more costly, area-wise. The speculative scheme is more complex design-complexity-wise (to a first approximation…).

Terms

BRU: Block **R**AM **U**nit, or the number of block RAMs required to store the smallest (but same) amount of PS and LS, taking into account the decoupled LS/PS width.
Graph fragment: The part of a graph derived from a single local order.
GS: Graph sampler.
LS: Local score.
MCU: MCMC controller unit.
MSU: MCMC Scoring network.
PPN: Parent sets per node.
PPC: Parent sets per core.
PS: Parent set.
OS: Order sampler.
SC: Scoring core.
SN: Scoring node.
ST: Scoring tile.
{SC,ST,SN,MST}_count: The number of SCs,STs,etc supported by the system. Each of these properties are known statically by any part of hardware that needs this information.