2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems

A Generic Implementation of Barriers using Optical Interconnects

Sandeep Chandran, Eldhose Peter, Preeti Ranjan Panda, and Smruti R. Sarangi
Department of Computer Science and Engineering,
Indian Institute of Technology Delhi (IITD),
Hauz Khas, New Delhi – 110016, India
{sandeep, eldhose, panda, srsarangi}@cse.iitd.ac.in

***Abstract*—Barriers have long been recognized as important performance-critical constructs in parallel applications. As a consequence, researchers have proposed fast implementations of barriers in both traditional electrical networks and in non-conventional networks such as optical NoCs. We prove in this paper that current protocols for barriers in optical NoCs are simplistic and cannot be trivially extended to accommodate for normal events that arise in regular operation such as presence of multiple applications, context switches, thread migrations, and variability in the number of active threads. We propose two generic protocols for barriers that can take all such cases into account, are fast, and try to minimize the number of messages sent over the NoC. One of these protocols is a centralized protocol (suitable for less cores), and the other is a distributed protocol, which is scalable. For a suite of standard benchmarks we found the latter to yield a mean speedup of 30.77% over a design that uses a hardware tree barrier. Our barrier implementation per se is roughly 2X and 20X faster than prior implementations that use transmission lines and electrical links respectively.**

I. Introduction

Due to continued scaling as predicted by the Moore's law, we shall have large multicore chips with at least 32 to 64 cores very soon. Moreover, over the next 5-10 years, we expect to have 128-256 core chips. It will then be possible to run high performance computing (HPC) applications on such manycore chips. At the moment such applications are being run on clusters of servers. However, before the large scale adoption of such HPC applications on manycore processors, it will be necessary to implement some standard performance enhancing features that are already present in high performance clusters. One of the most important such features is support for barriers. A *barrier* is defined as a rendezvous point for multiple threads. It is most commonly used to start reduction operations after a set of threads have completed their tasks. Currently, HPC clusters have either dedicated networks for barriers, or use fast software implementations of barriers because it is widely recognized that barriers are critical to performance. In manycore processors researchers have also realized the importance of barriers and have thus proposed protocols [1], [2], [3] for implementing fast barriers.

The problem of implementing barriers in traditional electrical networks has been widely studied. However, recently the community is aggressively looking at fast non-conventional on-chip interconnect technologies such as optical and RF interconnects. As compared to electrical NoCs (network-on-chips), such interconnects have significantly lower latencies (1-2 ns corner to corner), higher bandwidth, and naturally support broadcast based traffic. Some of these features such as the fast broadcast capabilities confer some unique advantages to designers for creating protocols for ultra-fast barriers. There is some preliminary work in this area [4], [5] that proposes methods for implementing barriers using RF and optical interconnects respectively. The basic approach is to use a broadcast bus to implement a wired-OR or wired-AND kind of logic. For example with optical NoCs, each core can transmit a signal at a predetermined wavelength on a broadcast bus, when it has reached a barrier. When a core receives signals from all the other cores (at their specified wavelengths), it can decide to proceed past the barrier. There are many alternative renditions of this protocol that we can design. For example, we can change the sense of the signals (transmit before the barrier, and then stop), or transmit a signal at only a single wavelength and a core can absorb it if hasn't reached the barrier.

In this paper, we identify a fundamental issue with such prior proposals, which are simplistic in our view. In specific, they assume that the number of cores is equal to the number of threads, every thread is interested in entering the barrier, there are no context switches or thread migrations, and we do not need to support multiple barriers at the same time. All of these are fairly restrictive assumptions, and do not hold for programs running in the field. It is hard to fathom a system that does not allow multiple applications to run in parallel, and assumes before hand or at compile time that we know about the behavior of all the threads in a program. Traditional methods to implement barriers with electrical NoCs also make some of these restrictive assumptions. However, it is easy to design methods to circumvent these special cases because the latencies involved in electrical NoCs with complex routers are large (50-100 cycles). For example, in typical hardware tree based barriers, we can change the structure of the tree if there is a context switch, or we can resort to a centralized protocol where all threads send their status to a central entity. There will be no correctness issues. With non-conventional NoCs (such as optical NoCs) the same set of options are available. However, the main benefit of implementing a barrier on an optical NoC, which is speed, needs to be sacrificed. It will take time to take corner cases into account, and we will not be able to achieve our original aim of having a very fast barrier implementation.

As a part of our work, we prove that current protocols cannot be trivially extended to consider all the cases that are possible during normal operation. There will be correctness issues. To implement a generic protocol for barriers, we prove (see Section III and online Appendix [6]) that we need to maintain state consisting of at least $\lceil log(N) \rceil$ bits (N is the number of threads), maintain and broadcast a unique barrier-id, and elect a leader (co-ordinator) among the threads if we wish to get a $O(N)$ times reduction in the number of messages that are sent. Keeping these requirements in mind, we design a new protocol that allows us to implement the general case, where a barrier allows an unbounded number of unknown participants, and is immune to problems caused by

978-1-4673-8700-2/16 $31.00 © 2016 IEEE
DOI 10.1109/VLSID.2016.16

thread migrations and context switches. We simulate both a centralized and distributed version of our protocol for a suite of standard benchmarks (Livermore Loops [7] and Parboil [8]). Our barrier release latency in the distributed protocol is 2 processor cycles (1 cycle=500 ps). We show that the mean speedup with the distributed protocol is 30.77% over the hardware tree barrier for a standard suite of Livermore and Parboil benchmarks. Our barrier implementation per se is faster (in terms of release latency) than state of the art implementations using transmission lines and electrical links by factors of 2X and 20X respectively. We designed and synthesized all our hardware units using industry standard tools, and showed that the area overheads are minimal (0.01% for a $400mm^2$ chip).

II. Background and Related Work

Basics of Barriers: A barrier is a software construct that allows multiple threads (in a barrier group) to synchronize. It has four attributes – (i) $capacity$ (number of threads in the barrier group), (ii) $count$ (number of threads that have currently reached the barrier), (iii) $sense$ (a boolean variable that switches from 0 to 1 and back with every barrier release), and (iv) $address$ (physical address that maintains a copy of the attributes). These attributes are initialized through the *barrier_init()* operation which can also be performed by a thread that is not a part of the barrier group. Each thread calls *barrier_wait()* when it reaches a barrier. When all the threads in a barrier group reach a barrier they can all be released simultaneously.

Optical Communication: We assume an off-chip laser source, and an on-chip SWMR (single writer multiple reader) network [9]. In such a network each transmitter is connected to all the other receivers using separate optical channels. It is a widely used network and seamlessly supports broadcast traffic. We can use any other type of optical network also. Our protocol is not dependent on the type of optical NoC.

Traditional Barriers: Sampson et al. [1] propose a hardware assisted barrier that bases its synchronization on the availability of cache lines. It takes hundreds of cycles. Sartori et al. [2] propose a Steiner tree based barrier that forms a least weight minimum spanning tree across the cores that are interested in a barrier. Threads send messages along the edges of the tree. In comparison, Stoif et al. [3] propose a centralized approach that uses a set of counters to keep track of the number of threads that have entered a barrier. To release a barrier, they use a fast NoC to broadcast the release event.

Barriers using Optical/RF NoCs: Barriers using novel technologies have proved to be an order of magnitude faster than barriers with traditional electrical NOCs. TLSync [4] proposes to implement barriers using an on-chip transmission line. Each processor in a barrier group transmits a signal in a known RF frequency on the shared transmission line before reaching the barrier. Once it reaches the barrier it stops transmitting. All processors know about the barrier release when there is no signal on the transmission line. The main issue with this paper is that the authors do not discuss how a frequency for a barrier group is allocated, and secondly they do not mention what happens if a thread gets swapped out, or a thread migrates to another core. This is a non-trivial problem and requires additional state and communication (see Section III). Binkert et al. [5] envision a barrier purely using an optical bus based network that makes two passes around the cores. In the first pass of the bus, they propose to transmit a signal at a prespecified wavelength along the optical bus and any core that has not reached the barrier needs to divert all the light. Once all the threads have reached the barrier no thread diverts the optical signal, and its presence can be deduced by sensing for the optical signal in the second pass of the bus. Allocating a specific wavelength, letting all the participating threads (which are unknown at compile time) know about it, and taking care of thread context switches are non-trivial problems, which have not been discussed in the original paper. Moreover, it is hard to extend the protocol to support simultaneous barrier operations by multiple processes because we are limited by the number of wavelengths (32-64) that the system can support.

III. Theoretical Results

Due to lack of space, we shall only state the main results of our theoretical contribution here. Readers can refer to the online technical report posted at [6].

Related work [4], [5] assumes that the states of all the barriers are available simultaneously, and it is possible to compute a Boolean function on them. Individual stations (transmitter/receiver) need not maintain any other state in memory. We prove that in the general case, where the number of threads is possibly more than the number of cores, we need to store at least $\lceil log(N - M) \rceil$ bits in memory, where N is the number of threads in a barrier group, and M is the number of threads that are scheduled on cores. It is always possible that M can be 0, and thus we need to store at least $\lceil log(N) \rceil$ bits in memory. Current techniques do not propose any method of incorporating additional state stored in memory. Second, current approaches send only single bit signals that indicate whether threads have reached the barrier or not. We prove that this is not sufficient, and when we have multiple barriers in the system, we will have correctness issues. It is necessary to broadcast a unique barrier-id at least once for each barrier entry/release cycle. Modifying [4], [5] to send multi-bit signals is non-trivial; it will necessitate changing the basic communication substrate. Finally, we also prove that it is necessary to elect a leader among the set of cores running the threads in a barrier group, if we want to minimize the number of messages. We show that if we have a leader (co-ordinator) we can reduce the number of messages by a factor of $O(N)$. Current approaches do not have a leader, and thus they need to transmit their status all the time leading to a lot of wasted laser power. By taking all of these constraints into account we created a generic protocol that is significantly different and sophisticated as compared to prior work.

IV. Implementation

We begin by appending each core with a small piece of hardware called the *barrier unit* which is responsible for handling all the barrier operations on behalf of the core, and it also maintains a local copy of the state of the barrier used by the current thread. The copy of the barrier-state is made a part of the context of the thread, and hence, is swapped in and out during context-switches. This avoids the need to maintain the barrier-state at the barrier units across context switches, thereby minimizing the overall area of the barrier unit. This also enables us to support a potentially unlimited number of simultaneous barriers.

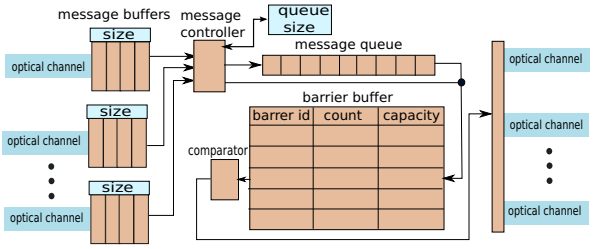

Fig. 1: Central Station (Centralized Protocol)

We assume a 48 bit physical address (similar to AMD64 architectures). Every barrier is uniquely identified by a barrier id (65 bits): 48 bit physical address, 16 bit process id, and the sense of the barrier (1 bit). It is important to note that the barrier-id remains the same across all the participating threads. Each message passed in our proposed architecture consists of 3 fields, (i) a 3 bit message id that identifies the type of the message, (ii) the 65 bit barrier id, and (iii) a 6 bit thread id/barrier count (depending on the message type). The memory address associated with a barrier stores its count and sense.

A. Centralized Protocol

We add a global structure called the *central station* that maintains the state of each barrier in the system, and is connected to all the cores via optical stations. During the execution of a barrier_init(), the thread sends a `REGISTER` message to the central station with the address of the barrier, and the capacity of the barrier. This creates a new entry in the *barrier buffer* inside the central station, along with initializing the barrier state maintained in the corresponding memory location. After initialization, as the threads enter the barrier by invoking *barrier_wait()*, an `ENTRY` message is sent to the central station to notify it. This can be done through either an I/O instruction or a write to a model specific register. The thread can then perform a spin lock on a dedicated register, or go to sleep. It is then woken up by the barrier unit, and allowed to proceed with execution upon the receipt of a `RELEASE` message sent by the central station, when the barrier is released.

Handling Context-switches: If a thread is swapped out before it can send its `ENTRY` message, we set a bit in its context. After it is swapped back, its barrier unit tries to send the `ENTRY` message. Secondly, it is possible that when a `RELEASE` message arrives, a given thread may be swapped out. To remedy this problem, each time a thread waiting on a barrier gets swapped in after a context switch, it checks if its local copy of the sense matches with the sense of the barrier in its memory location. In case of a mismatch, the thread releases itself. To take memory consistency issues into account, it is best if the thread issues a read request τ_w cycles after it has been swapped in. Here, τ_w is the maximum number of cycles, a memory write request takes to complete.

Hardware design: Figure 1 shows a high-level design of the central station. It contains a set of message buffers that buffer each incoming message for each input channel. The barrier buffer is implemented as a content-addressable memory (CAM) that is addressed by the barrier-id. The central station requires 2 cycles (@ 2 GHz) to process a message. The first cycle is required to lookup the CAM and access the entry in the barrier buffer. In the second cycle it increments the count and checks if the barrier can be released. We implement the central station as a 2-stage pipeline. The *message controller* feeds this pipeline either through the message queue, or from the message buffers directly.

Finally, it is possible that the number of simultaneous barriers exceeds the capacity of the barrier buffer. To handle such scenarios, we propose to add an overflow bit to the central station. If it is set to 1, then this indicates to the controller that additional barrier entries are stored in a dedicated region in main memory.

B. Distributed Protocol

```
   Data: barrier_t* barrier (address of barrier in memory)
 1 barrier_wait(mytid):
 2 begin
 3     < round1 > broadcast ENTRY message with tid
 4     < round2 > T2 ← ( tids of all ENTRY messages
       received)
 5     < round3 >
 6     T3 ← ( tids of all ENTRY messages received)
 7     received ACCEPT: busy_wait()
 8     received RELEASE: release()
 9     received no message: go to round 4
10     < round4 >
11     minTid ← min(T2 ∪ T3 ∪ mytid)
12     if minTid = mytid then
13         send ACCEPT message
14         count ← | T2 | + | T3 | + 1
15         isTrans ← 0, declare self as co-ordinator (see
           lines 18-22), and process messages
16     end
17 end
18 declare self as co-ordinator:
19 after τw cycles:
20 if localSense = barrier.sense then
21     count ← (isTrans) ? count: (count +
       barrier.count)
22 end
23 received RELEASE message / release():
24 localSense ← ! localSense
25 release barrier
26 context switch-in:
27 if waiting at barrier then
28     after τw cycles if localSense ≠ barrier.sense then
29         release()
30     else
31         sleep()
32     end
33 end
34 Co-ordinator: receive ENTRY message:
35 count ← count + #(ENTRY messages received)
36 if count = capacity then
37     send RELEASE message
38     barrier.sense ← !barrier.sense
39     barrier.count ← 0
40     release()
41 else
42     send ACCEPT message
43 end
```

In the distributed protocol, the decision of releasing a barrier is taken collectively by the participating threads. The

Co-ordinator: context switch-out:
$< round1 >$ send TRANSFER message
$< round2 >$ $\mathcal{T}_R \leftarrow$ all REPLY messages
$< round3 >$ **if** $\mathcal{T}_R = \phi$ **then**
  barrier.count $\leftarrow$ $count$
  barrier.sense $\leftarrow$ $localSense$
else
  $< round4 >$ send COUNT message with count
end
received TRANSFER message:
$< round2 >$ send REPLY message with tid
$< round2 >$ $\mathcal{T}_R \leftarrow$ tids of all REPLY messages
$< round3 >$ **if** *mytid = min(*$\mathcal{T}_R \cup mytid$ *)* **then**
  $isTrans \leftarrow 1$, declare self as co-ordinator
end
$< round4 >$ receive COUNT message, and set its value to $count$

count of the number of threads that have currently reached a barrier is maintained by a specific barrier unit elected as the **co-ordinator** by the participating threads. Every message sent by a barrier unit is broadcast to all the other barrier units. The barrier units that are not a part of the barrier group discard such messages.

Protocol: The distributed protocol is divided into several rounds. In our implementation of the protocol, a round spans m clock cycles, and it always starts at a cycle that is an integral multiple of m (m = 2 in our design). All the barrier units maintain a cycle count (without any skew). In the first round, a core sends an ENTRY message with the barrier id (65 bits), and its thread id (6 bits) to all the cores. In the second round, the co-ordinator (another barrier unit in the barrier group) increments the barrier $count$ with the number of ENTRY messages that it received. If the barrier count is equal to the capacity of the barrier, then the co-ordinator sends a RELEASE message in the third round. The co-ordinator proceeds to flip the sense of the barrier stored at the address associated with the barrier in the memory, and each participating thread flips its local sense. If the barrier cannot be released, the co-ordinator sends an ACCEPT message with the barrier id.

It is possible that there is no co-ordinator, or the co-ordinator has been swapped out. In this case, the cores do not receive the ACCEPT message in the third round. Each core computes the minimum thread id received in the second and third rounds. The core with the minimum thread id declares itself to be the co-ordinator in the fourth round (no need to send a message). It initializes the barrier count with the number of ENTRY messages received in the last two rounds. We defer processing context switch events till a barrier unit completes all its rounds, and all its outstanding memory read requests (if any) complete. Hence, we can safely assume that a barrier unit is not interrupted in the middle of processing a $barrier\_wait()$ event, or a message sent on the bus.

Handling Context-switches: Before a context switch, the co-ordinator needs to transfer its role to another barrier unit that is waiting for the same barrier, if there is one. It broadcasts a TRANSFER message with its barrier id, and barrier count. All the other units that have threads in the same barrier group, reply with their thread id, using a REPLY message in the next round. Each participating barrier unit computes the minimum thread id. If a certain barrier unit finds itself to be the new co-ordinator, then it initializes its state with the barrier count

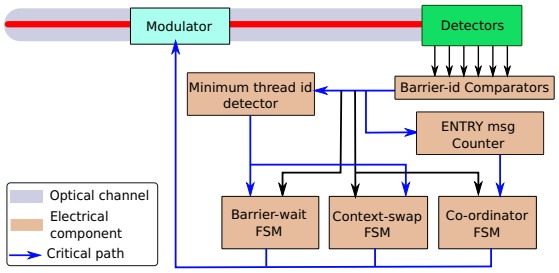

Fig. 2: Barrier Unit (Distributed Protocol)

equal to 0. The old co-ordinator sends its current count to the new co-ordinator in a COUNT message. If the old co-ordinator does not find a new co-ordinator, then it writes the barrier count and its local sense to a fixed location in shared memory (barrier address), and then performs the context switch.

It is possible that a set of threads enter the barrier, elect a co-ordinator, and then all of them get swapped out. Later on another set of threads enter the same instance of the barrier. The new co-ordinator needs to be aware of the barrier count recorded by the old co-ordinator. Hence, we add a new step after a barrier unit takes on the role of a co-ordinator. It initiates a read to the barrier address, and if the sense matches, it gets the value of the barrier count. It then adds the fetched value to its barrier count. Secondly, a thread might get swapped in, and then realize that it is waiting for a barrier. The barrier might have already been released. The barrier unit needs to initiate a read to the barrier address τ_w cycles after getting swapped in, and see if the barrier has been released (check if the sense has reversed).

Hardware design: The design of a barrier unit implementing the distributed protocol is shown in Figure 2. There are 3 separate controllers within the barrier unit. The *barrier_wait FSM* handles the barrier operations related to ENTRY and RELEASE. The *co-ordinator FSM* deals with electing a co-ordinator. The last controller, *context-swap FSM*, is responsible for handling context switches. These controllers co-ordinate among themselves based on the message type and the state of the thread. Since, detection of the minimum thread id is a crucial step when a co-ordinator is being elected during ENTRY and TRANSFER, we instantiate a separate block for it, which is shared between these controllers. The comparators are arranged as a tree in order to compute the minimum thread-id quickly. This increases the area overhead but limits latency. The modulators and detectors convert the signals from the optical to electrical domain and vice-versa.

We divide the duration of a round into two intervals. In the first interval, we separately aggregate the ENTRY and REPLY messages sent by other threads in the barrier group. In the second interval, we count the number of messages in each category, and also find the minimum thread id. For both of these operations, we use a tree of adders, and comparators respectively. At the lowest level, the tree of adders consists of 1-bit adders only. The width of the adders increases as we proceed from the leaves to the root of the adder tree. Each input bit at the lowest level indicates whether or not a message has arrived from the corresponding barrier unit. Each comparator inside the tree of comparators is 6-bits wide (width of the thread id) and computes the minimum of thread ids of all the nodes in its sub-tree. These are the slowest operations in

our distributed protocol, and determine the critical path. An astute reader might argue that we can have a similar unit in the central station (centralized protocol) to process messages in parallel. However, we need to have 64 such units, because we can have 64 barriers progressing in parallel (one per core). Secondly, we need dedicated logic to allocate and de-allocate these units across barrier groups. The resulting complexity, area and timing overheads are prohibitive.

V. Results

A. Overheads

The main objective of this paper is to design hardware to support multiple barriers, unknown participants, and context switches. As discussed in Section II we use a standard optical SWMR network similar to [9]. The extra network traffic (in a system with only optical networks) due to the additional barrier messages is low ($< 5\%$) and we also do not propose any extra waveguide for this implementation. Due to the low message overhead, the additional power requirement is minimal. However, we do require that all the clocks in the barrier controllers be synchronized with the optical bus clock. Implementing this is not difficult because the barrier controllers can be in the same clock domain as the optical stations, which with today's technology typically need to run at the same frequency as the bus clock. Readers can refer to the paper on synchronizers for optical networks by Ortin-Obon et al. [10].

B. Setup

We simulate the timing of the core, network on chip (both optical and electrical) and memory systems using the cycle accurate Tejas architectural simulator [11]. Table I shows the configuration of the simulated 64 core system. Table IV shows the optical parameters of the simulated interconnect. We synthesize the central station, and the barrier units for both the protocols using the Cadence Encounter RTL compiler with the 90nm UMC standard cell library. The results are scaled to 22 nm using the results in [12]. For the centralized station, we use 4 message buffers per input channel, 16 channels, and a 32 entry barrier buffer.

We simulate 5 barrier intensive benchmarks (similar to prior work [4], [1]). *Livermore_1* and *Livermore_3* kernels are parallel loops from the Livermore suite [7]. *Matrix_mult* is an implementation of a parallel algorithm for matrix multiplication. *Monte_carlo*, and *Steady_heat* are heat flow benchmarks that were developed in-house using POSIX threads based on the algorithms given in [13]. For the purpose of fair comparison, we run the rest of the system using electrical networks, and use the optical/RF network only for barriers.

In our experiments we set the number of threads equal to the number of cores, and compare the performance of our barriers with other barrier implementations. None of the other hardware based barriers explicitly take context switches into account. Neither do they propose methods to consider the associated correctness and performance issues. Hence, we did not deem it appropriate to compare our results with other barrier implementations in the presence of context switches.

C. Barrier Latency and Performance

Figure 4 shows the mean release latency of a barrier across a range of hardware and software implementations. We always keep the die size constant at 400 mm^2 according to ITRS [14] projections. The tournament and sense reversing barriers are the most scalable software implementations. Even beyond 4 cores, they start taking an excess of 1000 cycles. In comparison, the hardware tree barrier is limited to 120 cycles for a 128 core system. TLSync350 is a fast transmission line based barrier [4], and it requires 10 cycles across all the configurations. Without contention, our distributed protocol requires 3 rounds (6 cycles), and the centralized protocol requires (4 cycles).

Figure 3 shows the performance improvement of the benchmarks relative to a HW tree barrier. The speedups of the distributed protocol range from 2.94 to 85.21%, with a mean of 30.77%. The speedups of the centralized protocol range from 1% to 19.9%, and a mean speedup of 8.19% (due to contention). The increase in performance depends primarily on the number of instructions executed by a thread between successive barrier invocations (shown in Table II). As the number of instructions between successive $barrier\_wait$ calls increases from 245 (in *Livermore_1*) to 8129 (in *Steady_heat*), the observed speedup dips sharply. Essentially, the performance becomes less sensitive to the latency of a barrier.

Let us look at Figure 5 to study the effect of contention among the participating threads. It shows the number of cycles taken by *Livermore_1* for 300 iterations when the number of threads is varied from 4 to 128. We observe that the number of cycles increases with an increasing number of participating threads in the case of the centralized protocol, but remains constant in the case of the distributed protocol. This is because the centralized station is designed to have a queue that contains messages possibly coming for different barrier groups. Subsequently, the messages are sorted by barrier group, and the state is updated. The process of queueing, sorting, and updating the state by accessing the *barrier buffer* is a sequential bottleneck. Whereas, the co-ordinator in the distributed protocol handles messages for a single barrier group, and can thus process messages in parallel without any inspection and buffering.

D. Synthesis results

The modulators and detectors that convert a signal from the electrical to optical, and the optical to electrical domains take 100 ps each [15]. The maximum length of an optical waveguide connecting 16 optical stations on a $400mm^2$ die is $50mm$ in the modified SWMR bus topology. Therefore, the maximum propagation delay is 350 ps. Hence, it takes 550 ps for an optical broadcast.

The area of the synthesized central station (see Table III) is 19616 μm^2 (34% combinational logic, 64% sequential logic). The barrier unit of the centralized protocol occupies 156 μm^2 and has a delay of 40 ps. The total area requirement for the centralized protocol is: $19616 + 156 \times 64 = 29,600$ μm^2. The area occupied by a distributed barrier unit is 1091 μm^2 (total: 1091×64 = 69,824 μm^2), and the length of its critical path is 366 ps. The comparatively low duration of the critical path is because the operation of the tree of adders and the tree of comparators is mutually exclusive. There is no round where both the trees are active. Thus, the total end to end delay including the optical signal propagation, modulation, and demodulation time, is 916 ps (which is less than the duration of a round).

TABLE I: Processor parameters

Architectural Parameters	
Frequency/ Die Size	2GHz/ 400 mm^2
Cores	64 (16 clusters)
Technology	22nm
NOC	mesh (XY routing)
Fetch /Decode /Issue Width	4/4/4
ROB Size /IW Size	168 / 54
Int Reg / Float Reg	32/32
L1 size (per core)	16KB
L2 size (per cluster)	2 MB
Associativity	1(L1)/ 8(L2)
Latency	2(L1)/ 22(L2)
Memory Controllers	1(per cluster)
Memory Latency	200 cycles

TABLE II: Benchmark characteristics

Benchmark	#insts/barrier	IPC
Livermore_1	245	2.35
Livermore_3	661	2.51
Matrix_mult	694	2.39
Monte_carlo	2035	0.73
Steady_heat	8129	2.55

TABLE III: Area and latency

	Area	Latency
Central station	19616 μm^2	452ps
Barrier unit (distributed)	1091 μm^2	366ps
Barrier unit (centralized)	156 μm^2	40 ps

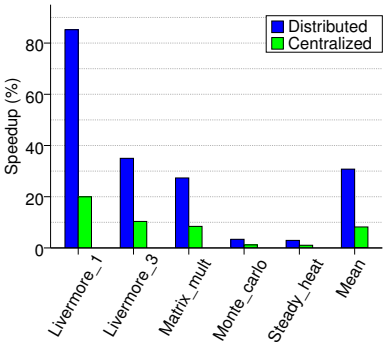

Fig. 3: Performance relative to the HW Tree barrier

TABLE IV: Optical parameters [15], [16]

Optical Parameters	
Wavelength (λ)	$1.55\mu m$
Width of waveguide (W_g)	$3\mu m$
Slab height	$1\mu m$
Rib height	$3\mu m$
Refractive Index of $SiO_2(n_r)$	1.46
Refractive Index of Si (n_c)	3.45
Combined transmitter and receiver delay	200 ps
Optical propagation delay	7 ps/mm
Electrical propagation delay	20 ps/mm

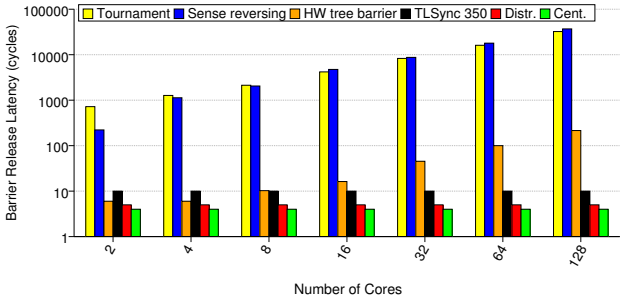

Fig. 4: Release latencies of software and hardware barriers

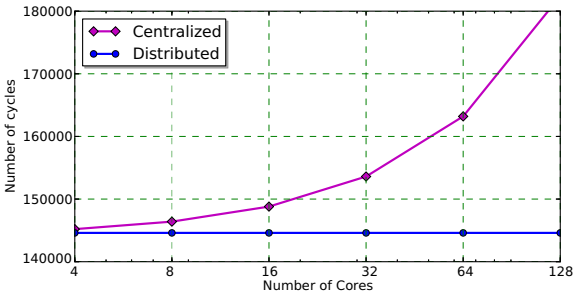

Fig. 5: Centralized vs. the distributed protocol

VI. Conclusion

We have proposed and evaluated two broadcast-based, barrier protocols – centralized and distributed, which can handle unknown participants, context switches, and thread migrations. We use a next-generation SWMR optical bus as the broadcast interconnect in our implementations. We show that the distributed protocol scales very well as the number of participating threads increases and is also well-suited for applications that require frequent synchronization. We observe a mean speedup of 30.77% across a suite of 5 parallel applications relative to a state of the art HW tree barrier.

References

[1] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *MICRO*, 2006.

[2] J. Sartori and R. Kumar, "Low-overhead, High-speed Multi-core Barrier Synchronization," *HiPeac*, 2010.

[3] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware synchronization for embedded multi-core processors," in *ISCAS*, 2011.

[4] J. Oh, M. Prvulovic, and A. Zajic, "TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines," in *ISCA*, 2011.

[5] N. Binkert, A. Davis, M. Lipasti, R. Schreiber, and D. Vantrease, "Nanophotonic Barriers," in *Workshop on Photonic Interconnects & Computer Architecture (in conjunction with MICRO 41)*, 2009.

[6] S. Chandran, E. Peter, P. R. Panda, and S. R. Sarangi, "Fundamental results for a generic implementation of barriers using optical interconnects," 2015. [Online]. Available: http://arxiv.org/abs/1510.00220

[7] T. Peters, "Livermore benchmarks." [Online]. Available: http://www.netlib.org/benchmark/livermorec

[8] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[9] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang, and A. Choudhary, "Firefly: Illuminating Future Network-on-Chip with Nanophotonics," in *ISCA*, 2009.

[10] M. Ortín-Obón, L. Ramini, H. Tatenguem Fankem, V. Viñals, and D. Bertozzi, "A complete electronic network interface architecture for global contention-free communication over emerging optical networks-on-chip," in *GLSVLSI*, 2014.

[11] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*, 2015.

[12] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[13] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. Tata Mc-Graw Hill, 2003.

[14] ITRS, *International Technology Roadmap for Semiconductors*, 2011 (Accessed Feb 10, 2013), http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[15] M. Haurylau *et al.*, "On-chip optical interconnect roadmap: Challenges and critical directions," *Selected Topics in Quantum Electronics, IEEE Journal of*, vol. 12, no. 6, pp. 1699–1705, 2006.

[16] I. O'Connor, "Optical Solutions for System-Level Interconnect," in *SLIP*, 2004.