Diamond Rings: Acknowledged Event Propagation in Many-Core Processors

Stefan Nürnberger[1], Randolf Rotta[1(✉)], Gabor Drescher[2], Daniel Danner[2], and Jörg Nolte[1]

[1] Brandenburg University of Technology, Cottbus-Senftenberg, Cottbus, Germany
{snuernbe,rrotta,jon}@informatik.tu-cottbus.de
[2] Friedrich-Alexander University Erlangen-Nuremberg, Erlangen, Germany
{drescher,danner}@cs.fau.de

Abstract. Hardware and software consistency protocols rely on global observability of consistency events. Acknowledged broadcast is an obvious choice to propagate these events. This paper presents a generalized ring topology for parallel event propagation with acknowledged delivery. Implementations for various many-core architectures show increased performance over conventional approaches. Therefore, diamond rings are a prime candidate for implementations of distributed memory models.

1 Introduction

A central component of consistency protocols are low-latency, high-throughput notification mechanisms that guarantee global observability to the initiator [1]. For this purpose, we propose a new broadcast topology called *diamond rings*. Its design targets broadcasting of events that are rather small while the final acknowledgement of their propagation is crucial. The event processing may be postponed locally as long as the propagation goes on. Usually, larger data is just referenced by the events instead of being transmitted directly. It is the nodes' task to copy or update any additional data as necessary.

A prime example for such mechanisms are coherence protocols that provide multiple reader single writer (MRSW) access on memory locations. Before writing to a location, exclusive access has to be obtained for the writer by sending a request for ownership. Every node that has a valid copy must be informed of this request in order to invalidate their outdated copy and revoke any previous write access. The writer can proceed only after all other nodes have been notified and acknowledging the successful broadcast is therefore essential. A similar scenario are update protocols. A central node sequences the atomic data modification requests and then broadcasts the updates. The individual requests are complete once their update has reached all replica.

Low latency is clearly desirable for such broadcasts in order to reduce the stall time of writers [2]. Just as much desirable is high throughput in order to reduce the congestion when pipelining updates and independent ownership requests.

Broadcast topologies provide a trade-off between latency and throughput. The latency is caused by overheads along the longest path. Keeping it short

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 722–733, 2015.
DOI: 10.1007/978-3-319-27308-2_58

requires asymmetric topologies with a large number of communication partners per node [2], which introduces imbalanced overhead among the nodes.

The throughput is increased by pipelining multiple broadcasts and is limited by the single node with the highest processing overhead. Thus, topologies with balanced overhead and few communication partners per node are required. The highest throughput is achieved by rings with a single successor per node [3].

Ignoring the delivery acknowledgement, conventional balanced trees present a sensible trade-off between latency and throughput. However, the acknowledgements require additional reductions from the leaves to the tree root. In this scenario, the proposed diamond rings provide better throughput and latency than conventional balanced trees. In comparison to asymmetric topologies, they provide better throughput in exchange for slightly worse latency.

Section 2 reviews related work. The new diamond rings are introduced in Sect. 3 and compared qualitatively against conventional topologies. The broadcast mechanism was implemented for a variety of many-core architectures as discussed in Sects. 4 and 5 compares them quantitatively.

2 Related Work

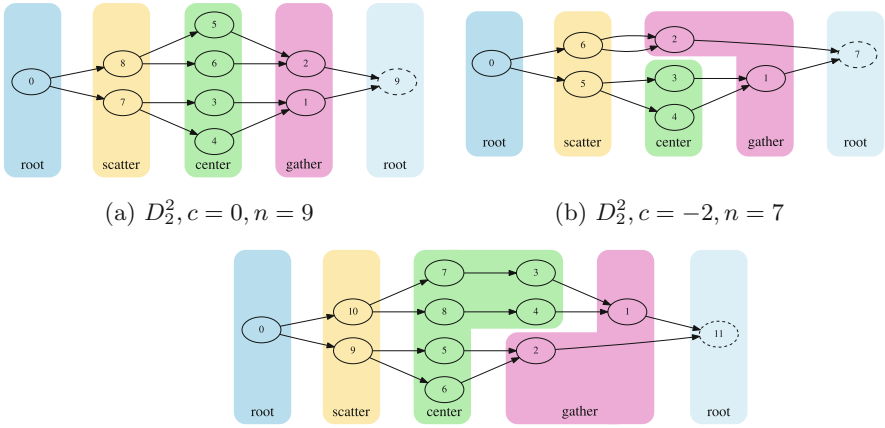
Efficient broadcasts received a lot of research, which can be divided into three categories: theoretical foundations, software-, and hardware-implementations.

The theoretical work focuses on finding optimal broadcasting trees for specific models of computation and communication networks. [4] summarises early work for very simple models of communication that ignore, for example, the message latency The most prominent enhanced models are POSTAL [5] and LogP [6]. The POSTAL model considers communication latency and simultaneous send/receive operations. The LogP model extends this by incorporating processor overhead, communication bandwidth, finite network capacity and multi-port I/O. Optimal broadcast algorithms in the LogP model were presented by [2].

Implementations face additional challenges like varying network latency and the efficient computation of optimal topologies at runtime [7]. Minimal-height lopsided trees can be constructed to cope with mixed-latency networks [8]. Other works focus on specific network architectures. For instance, [9] evaluates different broadcast algorithms on the Intel SCC [10] processor. They show that exploiting the SCC's 2D mesh with flat trees performs better than naive balanced trees. Finally, [3] points out that rings provide even better throughput.

Hardware-based approaches [1,11,12] optimise broadcast operations by dedicated support on the router and switch level. In practice, they apply the same ideas and topologies as software-based algorithms but have to cope with stricter resource constraints. Likewise, our proposed diamond ring topology can be applied in soft- or hardware and requires just point-to-point communication.

To the best of our knowledge, none of the previous research considered broadcasts with acknowledgement as a combined operation and the trade-off between latency versus throughput is often neglected. Research focused on the broadcast latency alone or, as in [13], only on the acknowledgement path.



(a) $D_2^2, c = 0, n = 9$ (b) $D_2^2, c = -2, n = 7$

(c) $D_2^2, c = 2, n = 11$

Fig. 1. Diamond Rings of arity 2 with different values of contamination. The two root nodes are conceptually the same but are drawn separately for clarity.

3 The Diamond Ring Structure

Based on the observation that the ring topology provides the best throughput but worst latency [3], we propose a generalisation towards parallel rings. This should trade loosing some of the throughput for a significantly reduced latency.

The *diamond ring* is a directed graph D_k^l with $|V(D_k^l)| = n$ nodes. The overall shape is a k-ary balanced tree of high l with the leaves merged to a mirrored tree as shown in Fig. 1a. It consists of four classes of nodes, called *root*, *scatter*, *center*, and *gather* nodes. The roots the two trees are merged into one root node. Each of the k^l center nodes is part of exactly one ring and the root is the only node that lies on all k^l rings.

This topology has a couple of advantages for the implementation of low overhead broadcasts. The path length is bounded by $O(\log n)$ due to the construction based on trees. The per-node memory requirement is $O(k)$ because memory is needed for each of the maximal k neighbours.

3.1 Extending to Arbitrary Node Counts

In practice, the desired node count does not match the size of a pure D_k^l. In such cases, c additional nodes must be introduced or removed beyond its regular topology, turning the graph into what we call a *contaminated* diamond ring.

The number of surplus nodes never exceeds the breadth of the ring center. Therefore, all potential modifications can be accomplished by inserting or removing up to k^l center nodes. With the center being the part exhibiting the highest degree of parallelism, this also implies that the ring's length is never increased by more than 1.

When deciding for the proper l for a given node count n, if $n > |V(D_k^l)| + k^l$ (i.e., more than k^l nodes have to be added), just choose D_k^{l+1} and remove some of the k^{l+1} nodes from the center. This always suffices, since the larger topology D_k^{l+1} features additional k^l nodes as counterpart to the old center nodes, as well as k^{l+1} new center nodes, i.e.

$$|V(D_k^{l+1})| = |V(D_k^l)| + k^l + k^{l+1} \quad .$$

There are alternative ways to handle contamination. For example, one can only add further center nodes. This is beneficial since the removal of nodes in the larger graph leads to multiple connections between the last level scatter and first gather node. However, the length of the longest path is also increased in this scheme. Figure 1 shows some examples for D_2^2 with different values of contamination.

3.2 Numbering and Addressing Scheme

A node is able to determine its neighbours solely by using its own node ID, the graph arity, and number of nodes in the topology. Beyond that knowledge, no further communication between the participating nodes is needed for topology setup. The parameters l (level, or depth) and c (contamination) are determined as $min\{l \mid -k^l < c \leq k^l\}$, where $c = n - |V(D_k^l)|$.

The node class (scatter, gather, etc.) is then determined by (1). The neighbours of a node are determined by (2) for a D_k^l with $c = 0$. The equations derive directly from the number of nodes in a balanced tree. Considering contamination requires some special cases, which are not shown here for brevity. Also omitted is the computation of offsets for root and gather nodes that receive messages from multiple predecessors, which requires basically a residue check. This is needed when disjoint message buffers or queues are used for the k predecessors.

$$type(id) = \begin{cases} \text{root node}, & \text{if } id = 0 \\ \text{scatter node}, & \text{if } \frac{n+k^l+1+c}{2} \leq id \\ \text{center node}, & \text{if } \frac{n-k^l-1}{2} < id < \frac{n+k^l+1+c}{2} \\ \text{gather node}, & \text{otherwise} \end{cases} \tag{1}$$

$$neighbours(id) = \begin{cases} \{n - i \mid i = 1..k\}, & \text{if } type(id) = \text{root} \\ \{n - (k(n - id) + i) \mid i = 1..k\}, & \text{if } type(id) = \text{scatter} \\ \{(id - 1)/k\}, & \text{if } type(id) = \text{center} \\ \{(id - 1)/k\}, & \text{if } type(id) = \text{gather} \end{cases} \tag{2}$$

3.3 Comparison to Tree-Based Broadcast with Reduction

One advantage of diamond rings over broadcast trees is the significantly reduced number of messages sent. Considering the center of a diamond ring with k^l nodes, the balanced tree requires exactly one more level to reach all nodes in the graph

before beginning the reduction. Together with the first reduction step to get back to the diamond ring's center, it adds up to $2k^l$ additional messages sent for the tree based reduction. The total number of nodes in a broadcast tree is $k^{l+1} - 1$. That means twice the number of messages for binary trees ($k = 2$). The factor decreases with larger k. With the same reasoning, it becomes clear that the longest path from the root, through all nodes, and back to the root is exactly two hops shorter for diamond rings.

The most important property is the reduced workload on inner nodes. Each node is an active part of the diamond ring broadcast exactly once. In contrast, balanced trees require the inner nodes to forward the messages *and* the acknowledgements. Hence, they are active twice for a single broadcast. In the diamond ring this is the case only for the root node, sending out a request and later receiving the acknowledgement. Inner nodes in the balanced tree engage in $2(k + 1)$ active communications either sending or receiving a message.

However, there is also a drawback for diamond rings in comparison to balanced trees regarding latency. In the tree broadcast the first thing a receiving node does is forward the message. Then it will continue with the required work before handling the acknowledgement. In a diamond ring, all work that needs to be acknowledged must be performed before the message is forwarded to the neighbours. Depending on the amount of work that is required for each message, latency may be worse for the diamond rings despite the marginally shorter hop count. If message reception (i.e., observability) is the only criterion that needs acknowledgement, forwarding may take place immediately. In a pipelined scenario, this increased latency does not affect overall throughput.

3.4 Root Node Overhead

In the proposed diamond ring, the root node sends messages to k different neighbours and receives k messages. In comparison, all other nodes communicate with at most $k + 1$ neighbours. This constitutes a throughput bottleneck, which can be alleviated quite easily.

An additional gather node can be introduced as companion to the root. It takes the position of node n in Fig. 1 and forwards the acknowledgement as a single message to the root. The path length thereby increases by one, which increases the latency slightly. Both, root and helper node, can issue acknowledged broadcasts. All nodes would have at most $k+1$ communication neighbours, thus eliminating throughput bottleneck.

4 Implementation Notes and Benchmark Variants

The diamond ring topology as introduced in the previous section defines the basic communication scheme. It can be implemented in very different communication models ranging from shared memory to hardware-based message passing. As a first step we implemented a simple task-based framework targeting Tilera, Xeon, and XeonPhi processors. The next section details this framework. Then,

the three evaluated broadcast variants *diamond rings* (DR), *sequenced diamond rings* (SDR), and *balanced trees* (BT) are presented.

4.1 Communication and Task Scheduling

The three target many-core architectures provide a Linux environment. To simplify porting, we implemented the framework as multi-threaded C++ application. The POSIX threads API is used to create one application thread per hardware thread (or core) and bind them via thread affinity. All threads operate in the same cache-coherent shared memory, which is used for communication.

In order to be able to compare throughput effects, a large number of concurrent broadcasts needs to be pipelined. Thus, each thread has to manage several overlapping activities. These are represented by task objects akin to active messages and contain a function pointer and additional payload for this function. Each thread has a thread-local LIFO task scheduler based on a double-ended queue from the C++ STL. For asynchronous inter-thread communication, each thread owns a multi-producer singe-consumer FIFO queue. The scheduler polls this queue when no local tasks are available or when send operations are stalled due to congestion. Received tasks are enqueued to the back of the local dequeue.

On the Xeon and XeonPhi processors, the communication queues were implemented as fixed-size ring buffers. Just pointers to statically allocated tasks residing in shared memory are communicated in order to avoid dynamic memory management. The Tilera architecture has a low-latency point-to-point network called UDN. It can be used to send small messages directly from one core to another and the intermediate network behaves as a FIFO queue. Our implementation uses the UDN for inter-thread communication by sending task pointers.

This is a quite general framework providing more than the simple broadcasts might require. However, actual applications usually need more than just broadcasts. In our experience, over-simplified broadcast communication channels tend to not cooperate well with the higher-level communication and scheduling.

4.2 Benchmark Variants

The following three benchmark variants were implemented for the evaluation as presented in the next section.

Balanced Trees (BT). The reference implementation is based on a balanced n-ary tree topology. Upon receiving a broadcast message, each node sends the broadcast message to its children and, then, processes the broadcast event locally. An acknowledgement message is sent back to the parent node after the local processing is completed and acknowledgement messages were received from all children. A separate node-local acknowledgement counter is used for each broadcast to track the outstanding acknowledgements at each node.

Diamond Rings (DR). The nodes operate differently depending on their position in the diamond ring. All scatter, center, and gather nodes begin processing the broadcast event on the first message received. They propagate the broadcast message to their successors after the local processing is complete and the broadcast message was received from each predecessor. Scatter and center nodes have just a single predecessor. The root node first sends the broadcast to its successors and, then, processes the event itself. The broadcast is completed at the root node after the local event processing completed and the broadcast message was received from each of the root's predecessors. Again, a node-local counter is used by gather nodes and the root to track outstanding messages.

Sequenced Diamond Rings (SDR). Tree and diamond ring topologies do not mandate any ordering of concurrently propagated events. The communication layer and the node-local task scheduling can reorder their messages and tasks. However, many application scenarios like, for example, request for ownership and distributed atomic updates require a strict ordering according to the event sequence at the root node.

The sequenced diamond ring implementation enforces in-order processing of pipelined broadcasts at each node. For this purpose, the root node assigns a sequence number to each event. A node-local sequence counter is used to delay broadcast tasks that are out of sequence. The sequencing can be implemented orthogonal to the broadcasts but a combined implementation was chosen to exploit cross-cutting optimisation opportunities.

5 Evaluation

This section compares the latency and throughput of acknowledged event broadcasts based on diamond rings against balanced trees. Trees are commonly used to propagate events and collect acknowledgements. Balanced trees were chosen because they achieve better throughput than skewed/asymmetric trees.

Based on the analysis in Sect. 3.3, following hypotheses are examined: (1) The longest path is shorter in diamond rings than in balanced trees. Hence, diamond rings should have a slightly lower latency as long as processing the broadcast event itself costs no time. (2) The balanced tree nodes have to process more messages than diamond ring nodes. Hence, the latter should provide higher throughput. In consequence, diamond rings would provide a better trade-off between throughput and latency than balanced trees and tree in general.

In order to evaluate these hypotheses, latency and throughput were measured in micro-benchmarks without actual application-level event payload. The latency is the time needed to complete individual broadcasts on otherwise idle cores. The throughput is measured with bursts of up to 128 pipelined broadcasts. In contrast to application benchmarks, this approach allows to study the performance differences in isolation.

The next subsection presents individual results for the three evaluation architectures and the last subsection discusses the overall results.

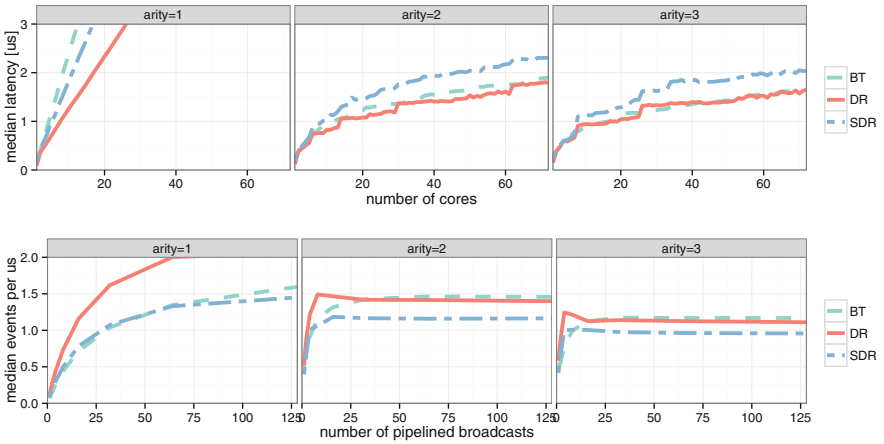


Fig. 2. Median latency and throughput on Tilera TILE-Gx72.

5.1 Benchmark Results

The measurements are carried out on the Tilera TILE-Gx72 and Intel Xeon-Phi 5110P many-core processors. For comparison, a large multi-core Intel Xeon machine is included. Both many-core architectures are based on in-order execution cores whereas the multi-core utilises out-of-order execution.

72-Core Tilera TILE-Gx72. This many-core processor contains 72 three-issue in-order VLIW cores running at 1 GHz. The cores are interconnected through several 2D mesh networks. The benchmark implementation uses Tilera's low-latency user-dynamic network (UDN) to communicate the pointers to messages and shared memory to access the message contents.

Figure 2 shows the latency and throughput results for arity 1, 2, and 3. All measurements were repeated 100 times. The mapping of topology nodes to cores was not specifically optimised. The highest throughput was achieved with arity 1 and diamond rings. With arity 2, the throughput is similar for diamond rings and balanced trees and diamond rings have a slightly better latency.

On this processor, processing overheads seem to dominate the communication overhead and latency significantly. The additional processing needed to order pipelined broadcasts in the sequenced diamond rings increases the latency compared to un-ordered balanced trees while the throughput is similar.

4x10-Core Intel Xeon E5 4640v2. This machine consists of 4 processors with 10 out-of-order cores per processor and two hardware-threads per core running at 2.2 GHz. The processors are connected through a cache-coherent QPI network. The communication is implemented via shared memory using a multiple-producer/single-consumer ring-buffer.

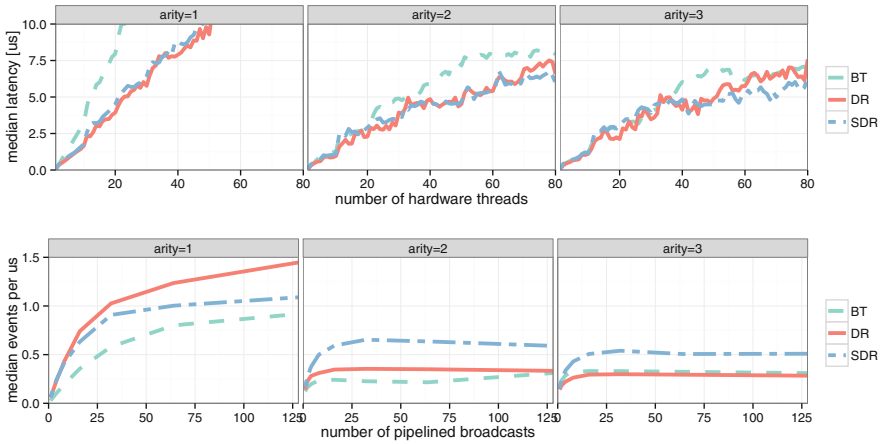


Fig. 3. Median latency and throughput on Intel Xeon E5 4640v2.

Figure 3 shows the results for this machine. Again, the highest throughput was achieved with arity 1 and diamond rings. With arity 2 and 3, the throughput of sequenced diamond rings the best while the latency is similar for all three.

Surprisingly, the sequenced diamond rings perform best, which could benefit from two characteristics of the architecture: The cores are designed for good single-thread performance, which compensates the additional processing overhead; And the enforced ordering might reduce the pressure on the limited bandwidth of inter-processor QPI links.

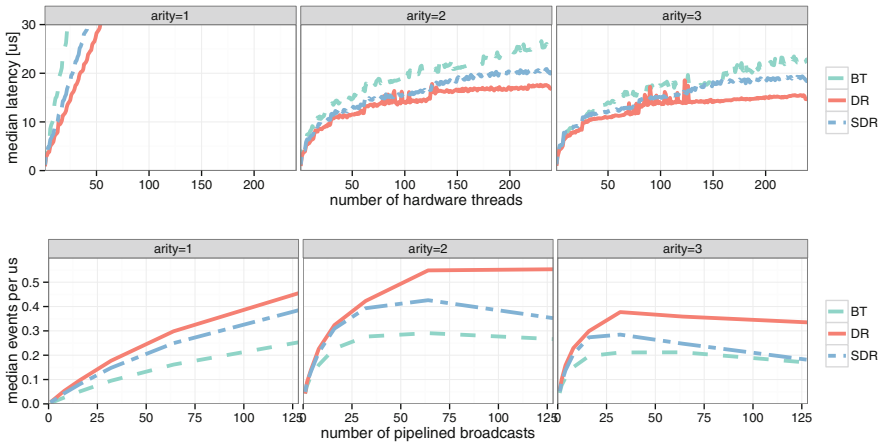


Fig. 4. Median latency and throughput on Intel XeonPhi 5110P.

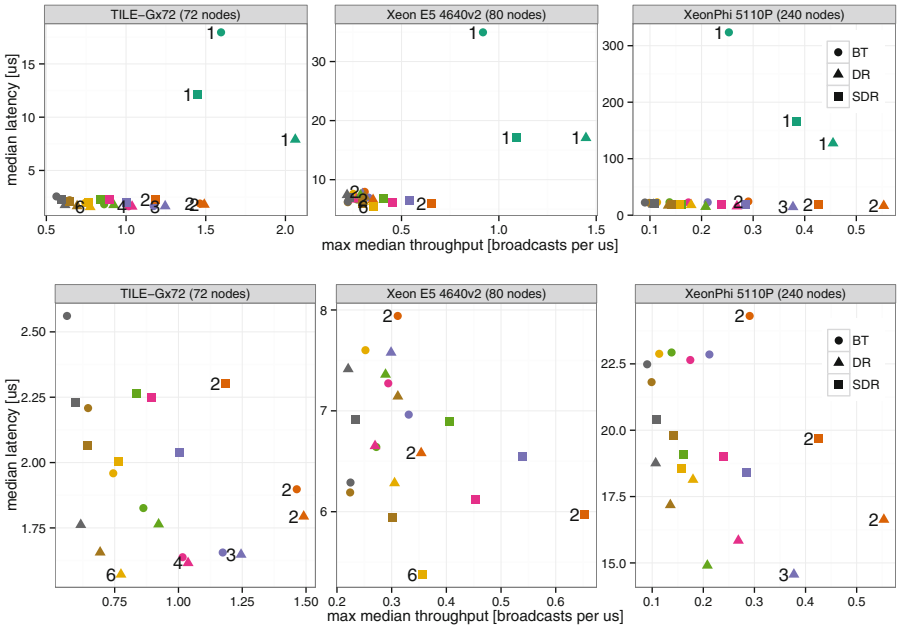


Fig. 5. Comparison of the single-broadcast latency against the largest observed throughput with pipelined broadcasts. Both figures show the same dataset but the lower figure excludes arity 1 for better readability. The arity is encoded by color and is, in addition, written left to interesting points. The better variants are to the lower right corner. Please note the different scales.

60-Core Intel XeonPhi 5110P. This many-core processor contains 60 in-order cores with 4 hardware-threads per core running at around 1 GHz. The cores are connected through two bi-directional rings. The benchmark uses the same ring-buffer implementation as above.

Figure 4 shows the results for this machine. The best throughput was achieved with arity 2 instead of 1. This is probably due to the arity 1 pipeline being much longer (240 stages) than the number of pipelined broadcasts. Diamond rings achieved better throughput and latency than balanced trees.

Ramos and Hoefler [14] presented an optimal design for small broadcasts and reductions on the XeonPhi processor using dedicated communication structures. For 60 cores, they report 10 µs latency for broadcasts plus 10 µs latency for reductions, i.e. acknowledgements. In comparison, the balanced 2-ary tree (24 µs) and 2-ary diamond ring (15 µs) presented here perform quite well while reaching all 240 threads and using a more general task-based model.

5.2 Discussion of the Results

Figure 5 compares the trade-off between latency and throughput directly for a larger selection of tree arities. The x-axis shows the peak median throughput and the y-axis the median latency for each benchmark variant.

The first column shows the TILE-Gx72. The Pareto optimal variants, beginning with lowest latency, are diamond rings with arity 6 (1.6 µs, 0.7 per µs), 4, 3, 2 (1.8 µs, 1.5 per µs), and finally diamond rings with arity 1 (8 µs, 2 per µs). For most arities, diamond rings performed better than balanced trees, which performed better than sequenced diamond rings.

The second column shows the results for the multi-core Xeon. The Pareto optimal variants, beginning with lowest latency, are sequenced diamond rings with arity 6 (5.4 µs, 0.35 per µs), then arity 2(6 µs, 0.65 per µs), and finally diamond rings with arity 1(17 µs, 1.45 per µs). For most arities, sequenced diamond rings performed better than diamond rings, which performed better than balanced trees.

Finally, the third column represents the many-core XeonPhi. The Pareto optimal variants, beginning with lowest latency, are diamond rings with arity 3 (15 µs, 0.37 per µs) and finally arity 2 (17 µs, 0.55 per µs). For most arities, the diamond rings performed better than sequenced diamond rings, which performed better than balanced trees.

On all three architectures, diamond rings achieved a higher throughput than balanced trees. The latency can be reduced by using larger arities. However, the latency does not decrease much beyond arity 2 while the throughput degrades quickly. In conclusion, diamond rings and sequenced diamond rings with arity 2 are a good choice for acknowledged event delivery.

6 Conclusions

We have presented a novel topology for efficient acknowledged broadcast to be used in memory consistency protocols. By combining the advantages of low latency in tree-based topologies and the high throughput achieved in ring-shaped communication, our diamond ring topology balances the overall utilization of the network and resource requirements on the participating nodes. We have implemented diamond rings on a multitude of platforms and compared their performance to existing approaches. Referring to the hypotheses made in Sect. 5, our evaluation results show that (1) the shorter path lengths in diamond rings result in lower latencies on all measured platforms, (2) the reduced computational load per node gives rise to higher overall throughput performance in diamond rings when compared to balanced trees. The choice of arity offers a trade-off decision between both those performance indicators, with higher arity reducing the latency at the cost of throughput. This shows that diamond rings constitute a prime candidate for use as an underlying communication layer in software memory consistency protocols.

Acknowledgments. This work was supported by the German Research Foundation (DFG) under grant no. NO 625/7-1 and SCHR 603/10-1. The evaluation on the Intel XeonPhi was supported by the German Federal Ministry of Education and Research (BMBF) grant no. 01IH13003C.

References

1. Jerger, N.E., Peh, L.S., Lipasti, M.: Virtual circuit tree multicasting: a case for on-chip hardware multicast support. In: ISCA 2008, pp. 229–240. IEEE (2008)
2. Karp, R.M., Sahay, A., Santos, E.E., Schauser, K.E.: Optimal broadcast and summation in the logp model. In: Fifth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993, pp. 142–153. ACM (1993)
3. Al-Khalissi, H., Bucty, R., Berekovic, M.: Efficient barrier synchronization for OpenMP-like parallelism on the intel SCC. In: Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, pp. 10–17, December 2013
4. Hedetniemi, S.M., Hedetniemi, S.T., Liestman, A.L.: A survey of gossiping and broadcasting in communication networks. Networks **18**(4), 319–349 (1988)
5. Bar-Noy, A., Kipnis, S.: Designing broadcasting algorithms in the postal model for message-passing systems. In: SPAA 1992, pp. 13–22. ACM (1992)
6. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., Von Eicken, T.: LogP: towards a realistic model of parallel computation. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1993, vol. 28, pp. 1–12. ACM, San Diego (1993)
7. Bruck, J., De Coster, L., Dewulf, N., Ho, C.T., Lauwereins, R.: On the design and implementation of broadcast and global combine operations using the postal model. IEEE Trans. Parallel Distrib. Syst. **7**(3), 256–265 (1996)
8. Golin, M., Schuster, A.: Optimal point-to-point broadcast algorithms via lopsided trees. Discrete Appl. Math. **93**(2), 233–263 (1999)
9. Matienzo, J., Jerger, N.E.: Performance analysis of broadcasting algorithms on the intel single-chip cloud computer. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2013), pp. 163–172. IEEE (2013)
10. Howard, J., et al.: A 48-core ia-32 message-passing processor with DVFS in 45nm CMOS. In: IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC 2010), pp. 108–109. IEEE (2010)
11. Malumbres, M.P., Duato, J.: An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors. J. Syst. Archit. **46**(11), 1019–1032 (2000)
12. Turner, J.S.: An optimal nonblocking multicast virtual circuit switch. In: 13th Proceedings IEEE of Networking for Global Communications, pp. 298–305. IEEE (1994)
13. Rothermel, K., Maihofer, C.: A robust and efficient mechanism for constructing multicast acknowledgement trees. In: Proceedings of Eight International Conference on Computer Communications and Networks, 1999, pp. 139–145. IEEE (1999)
14. Ramos, S., Hoefler, T.: Modeling communication in cache-coherent SMP systems - a case-study with Xeon Phi. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, pp. 97–108. ACM (2013)