WiDir: A Wireless-Enabled Directory Cache Coherence Protocol

Antonio Franques*, Apostolos Kokolis*, Sergi Abadal†, Vimuth Fernando*, Sasa Misailovic*, and Josep Torrellas*
**University of Illinois at Urbana-Champaign* †*Universitat Politècnica de Catalunya*
{franque2, kokolis2}@illinois.edu, abadal@ac.upc.edu, {wvf2, misailo, torrella}@illinois.edu

***Abstract*—As the core count in shared-memory manycores goes up, it is becoming increasingly harder to design cache-coherence protocols that deliver high performance without an inordinate increase in complexity and cost. In particular, sharing patterns where a group of cores frequently read and write a shared variable are hard to support efficiently. Hence, programmers end up tuning their applications to avoid these patterns, hurting the programmability of shared memory.**

To address this problem, this paper uses the recently-proposed on-chip wireless network technology to augment a conventional directory-based invalidation cache coherence protocol. We call the resulting protocol *WiDir*. *WiDir* seamlessly transitions between wired and wireless coherence transactions for the same data based on the access patterns in a programmer-transparent manner. In this paper, we describe the protocol transitions in detail. Further, an evaluation using SPLASH and PARSEC applications shows that *WiDir* substantially reduces the memory stall time of applications. As a result, for 64-core runs, applications take on average 28% less time to complete on *WiDir* than on MESI. Moreover, *WiDir* is more scalable than MESI. These benefits are obtained with modest area and power cost.

***Index Terms*—Multicore, Wireless NoC, Directory cache coherence protocol**

I. Introduction

Exploiting a combination of innovative chip manufacturing techniques and reduced semiconductor feature sizes, computer manufacturers continue to increase the core counts of processor chips. With more on-chip cores and bigger caches, systems can run bigger problems with limited cost increase. For example, Ampere's Altra [1] uses 7nm technology to support up to 80 ARM cores and a 32 MB Last-Level Cache (LLC) with a coherent mesh interface on a single die. As another example, an AMD Second Generation EPYC processor such as the 7742 [2] uses a chiplet organization and 7nm technology to support up to 64 cores (2-way SMT) and a 256 MB LLC with the Infinity Fabric interconnect. Further, an Intel Second Generation Xeon processor such as the Platinum 9282 [3] uses a dual-die package and 14nm technology to host up to 56 cores (2-way SMT) and a 77 MB LLC connected with a mesh interconnect. In the near future, it is likely that on-chip core counts will continue to increase.

For these manycores, shared memory is the most popular programming and execution model. The reasons are shared-memory's ease of programming, well-developed existing algorithms, and widespread libraries such as POSIX threads and OpenMP. To support shared memory at this scale, designers build directory-based hardware cache-coherence protocols (e.g., [4], [5]). Designing such protocols is an arduous process. Importantly, as the core count goes up, it is becoming harder to engineer cache-coherence protocols that deliver high performance without an inordinate increase in complexity and cost. This difficulty has been referred to as the *Coherence Wall*.

As an example, consider patterns where a group of cores frequently read and write a shared variable. Coherence protocols rely on invalidations to keep coherence [6]–[8], and do not support these patterns efficiently. As soon as a core writes the variable, all the other sharers get invalidated, and any subsequent read by another core suffers a costly cache miss. Sending invalidations and then re-reading the data creates long-latency transactions in a large on-chip interconnect. Update-based protocols [9] are not the solution either, as they have shortcomings for many common patterns, which has prompted designers to eschew them. As a result, if programmers want to attain high performance, they have to carefully tune the sharing behavior of their applications, ensuring that patterns like the ones considered appear infrequently.

Wireless on-chip networks (WNoC) is a new technology that has recently seen a lot of interest [10]–[17]. In WNoCs, each core or group of cores in a manycore has a transceiver and an antenna, which it uses to communicate with other cores. While the bandwidth of a WNoC is limited (only one or a few messages can be transmitting at a time), a WNoC supports low-latency transfers, since a message can cross a large manycore in about 5ns [18]. Further, WNoCs naturally support update multicasts and broadcasts. As a result, WNoCs have been proposed to speed-up applications on manycores. For example, WiSync [19] uses it to speed-up synchronization, while Choi *et al.* use it to accelerate CNN training in CPU-GPU heterogeneous architectures [20], and Replica [21] uses it to speed-up the execution of communication-intensive and approximate computations.

An intriguing question is whether a conventional, wired cache coherence protocol can be augmented to use a wireless network so that patterns like the ones described above can be supported efficiently. Frequent read-write sharing by multiple cores can indeed be efficiently supported by a WNoC: writes update all sharers without complicated multi-hop coherence transactions or lengthy message routing; reads can access data locally. However, to implement a complete coherence protocol, one needs to carefully combine wired and wireless transactions in a seamless manner.

This work was funded in part by NSF Grant No. CCF-1629431, and by EU's Horizon 2020 research and innovation programme Grant No. 863337 (WiPLASH).

In this paper, we present such a protocol, which we call *WiDir*. *WiDir* extends a directory-based invalidation cache coherence protocol with some wireless transactions. The goal is to efficiently support frequent read-write sharing within a group of cores — which has largely eluded conventional coherence protocols. *WiDir* seamlessly transitions between wired and wireless transactions for the same data based on the access patterns in a programmer-transparent manner. The end result is higher performance without the need to tune applications and hurt programmability.

For the design in this paper, we start with a conventional MESI protocol over a wired NoC. For a given cache line, *WiDir* uses this protocol when there are few sharers. Then, when the number of sharers goes over a certain threshold, the line transitions to the Wireless (W) state, where sharing between cores uses wireless transactions. When the number of sharers falls back down to the threshold, the line goes back to using MESI over the wired NoC. The paper describes the *WiDir* protocol transitions in detail.

Our evaluation using simulations running SPLASH and PARSEC applications shows that *WiDir* substantially reduces the memory stall time of the applications. As a result, for 64-core runs, applications take on average 28% less time to complete on *WiDir* than on plain MESI. Moreover, *WiDir* is more scalable than MESI. These benefits are obtained with modest area and power cost.

Overall, the contributions of this paper are:

• The novel use of WNoCs to enhance a cache coherence protocol. This use further builds the case for WNoCs.
• The *WiDir* directory-based hardware cache-coherence protocol that seamlessly combines wired and wireless transactions.
• An evaluation of *WiDir*.

II. Background & Motivation

A. Motivation for Wireless On-Chip Networks

Traditionally, wired on-chip networks (NoC) use a packet-switched interconnection fabric with each processor connected to a router as shown in Figure 1. Routers enqueue packets, compute routes, arbitrate, and dequeue packets in each hop towards the destination [22], incurring delays and energy consumption. To connect the routers, the mesh topology is the *de facto* standard due to its simple layout and low link length [22]. However, the average hop count scales proportionally to $\sqrt{N}$, where N is core count. There are other, high-radix topologies that reduce the network diameter [23], but at the cost of a non-trivial chip area and energy cost at the routers.

In contrast, WNoC architectures [10]–[17] are attractive because they can transfer a message chip-wide with a latency of only a few clock cycles, regardless of the size of the chip or the number of cores. Each core or group of cores is integrated with an antenna and a transceiver, as shown in Figure 1. Small antennas in the mmWave bands and beyond [24]–[27] can be feasibly integrated, and use a broadcast-based protocol [19], [21], [28]. Figure 1 shows vertical monopole antennas based on Through-Silicon Vias (TSVs), which perforate the bulk silicon [27]. Information coming from a core is modulated by the transceiver and sent by the antenna. The resulting signals propagate inside the chip package, bouncing off the metallic heat sink until they reach the receivers. Propagation causes signals to be attenuated a few tens of dBs, mainly due to spreading loss and the relatively high dielectric loss at the bulk silicon [29]–[31]. However, these losses are tolerable [18], and prevent the enclosed package to act as a reverberation chamber.

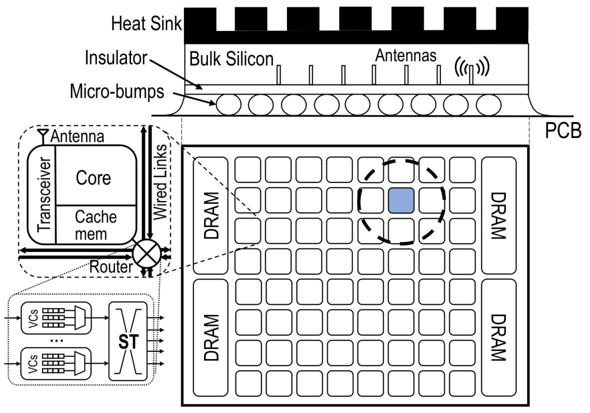


Fig. 1: Wired NoC mesh augmented with a wireless NoC, within a conventional flip-chip package with one vertical monopole antenna and transceiver per core.

B. Uses of Wireless On-Chip Networks

There are multiple proposals to use WNoCs to enhance the architecture of manycores [19]–[21], [28]. In general, these works leverage the low-latency and affordable broadcast of a WNoC to accelerate key patterns. Mondal *et al.* [28] and WiSync [19] use a WNoC to transfer all the accesses to synchronization variables, which are often traffic hot spots. In WiSync, synchronization variables are placed in a special memory called Broadcast memory. Using it only requires re-targeting the synchronization libraries or macros and, therefore, is transparent to programmers. Choi *et al.* [20] use a WNoC to transfer a fraction of the traffic generated during the training of a Convolutional Neural Network (CNN) in a CPU-GPU integrated architecture. The network design is architecture-aware and, like WiSync, transparent to the programmer. Replica [21] uses a WNoC to carry all the accesses to communication-intensive variables. The programmer has to explicitly identify these variables, which requires some effort. These variables are then placed in a special memory accessible by the WNoC. Replica also introduces approximate transformations, both in hardware and in software, which exploit the characteristics of WNoC communication. Mondal *et al.*, WiSync, and Replica partition the data structures into those that use the wired and those that use the wireless NoC, whereas Choi *et al.* do not.

In this paper, we focus on a different problem: enlisting both NoCs in supporting cache coherence protocol transactions. Both NoCs need to operate seamlessly together in a programmer-transparent manner.

C. Scalable Cache Coherence Protocols

Scalable cache coherence is attained through the use of directories [6]–[8], [32]–[34]. Directory cache coherence is used in commercial systems (e.g., [4], [5]). Such systems all use invalidation-based schemes [6]–[8] rather than update ones [9]. This is because, in general, update protocols create more traffic, and are subject to pathological cases, such as when data is left in a cache after a process migrates to another core, or when a process simply initializes data structures for several other processes, which will be running on other cores. There have been proposals for hybrid invalidation-update protocols [35].

In multiprocessors with large core counts, it is very costly for the directory to keep presence bits for all possible cores. As a result, designers use limited pointer schemes [7], [8]. When the number of sharers overflows the available pointers, a special action is taken, such as setting a broadcast bit, evicting a pointer, or re-configuring the directory entry.

The motivation for limited-pointer schemes is experimental data showing that an individual write often invalidates only a few caches [36]. But, as the machine core count increases, the average write invalidates more caches. Further, if the write did not invalidate the sharers, more sharers would accumulate, and a later write invalidate would be more expensive.

To gain insight into this issue, we modeled writes that update rather than invalidate, and measured the number of sharers that a line accumulates until the line is evicted from the LLC. For the 64-core machine and applications described in Section V, this number is on average 21 sharers. We then considered the cores that shared the line before a write, and measured what fraction of them re-read the line after the write. Such fraction is, on average, 56%. This data suggests that if, under some circumstances, we allow a write to perform a wireless update to the sharers rather than invalidating them, we may improve performance.

III. DESIGN OF WIDIR

A. Main Idea

As indicated before, in machines with large core counts, existing cache coherence protocols are unable to efficiently support groups of cores frequently reading and writing a shared datum. Indeed, state-of-the-art directory protocols for large core counts use invalidation-based limited pointer (or coarse-vector) schemes [4], [7], [8]. As more and more cores read the datum, the limited pointers overflow, and a special action is taken, such as setting a broadcast bit, evicting a pointer, or reconfiguring the directory entry. A subsequent write (and sometimes even reads) becomes more expensive.

Wireless communication is ideally suited to this type of sharing pattern. The writer core broadcasts the update to all the current sharers in a short, fast, multicast transaction. As the sharers issue subsequent reads, they obtain the up-to-date version of the datum from their caches.

In this paper, we present a hybrid directory-based cache coherence protocol that combines a wired and a wireless component. We call the protocol *WiDir*. The directory entry for any given shared line can dynamically transition between wired and wireless coherence transactions during program execution, depending on the current access pattern.

Initially, a line uses an invalidation-based wired protocol operating on the wired NoC. When the number of sharers for the line reaches a certain threshold, the line transitions to using wireless coherence transactions. In this mode, each sharer has a counter that counts the number of wireless updates received after every local read or write. If such count reaches a certain threshold, the sharer notifies the directory. Based on the number of such notifications, the directory can downgrade the line back to wired mode.

The *WiDir* protocol is supported by a manycore like the one shown in Figure 2. Each node in the manycore contains: a core with private caches (i.e., L1 instruction and data caches in the figure), a local slice of the shared last level cache (LLC) and its corresponding directory slice, a network interface, a local router to connect to the wired network, a transceiver, and two antennas to communicate wirelessly. To reduce costs in a large manycore, multiple neighboring nodes could share a single antenna pair.

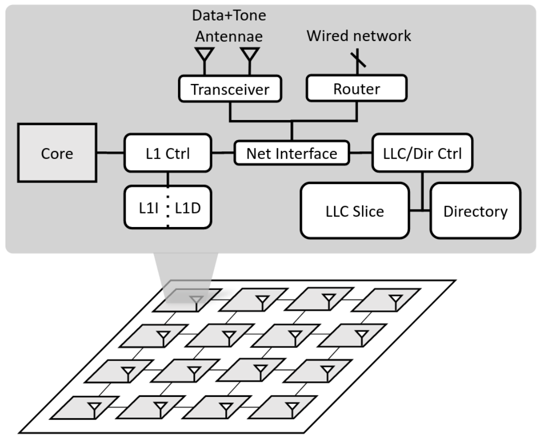


Fig. 2: Manycore that supports the *WiDir* protocol.

Following Abadal et al. [19], we use a data antenna and a tone antenna to communicate via a data channel and a tone channel, respectively. The data channel is centered at the 60 GHz frequency, and is used for data and most coherence messages; the tone channel is centered at 90 GHz and is used as a special-purpose *acknowledgment* channel. The data channel uses the BRS wireless protocol [37]. In this protocol, when a node has data to transmit, it listens to the medium. When the medium is free, the node starts transmitting, leaving the second cycle empty to give everyone a chance to report any detected collisions. If no collision is detected, the transmission continues to the end; otherwise, the transmission is squashed and restarted.

Cache coherence protocol messages issued by the local core or the local directory controller are passed via the network interface to the transceiver or the local NoC router, depending on the type of message that is being sent. For incoming messages from either of the two networks, the process is the same but in reverse.

B. Basic WiDir Operation

WiDir augments a vanilla invalidation-based directory coherence protocol with a new stable state, called *Wireless* or Wireless Shared (W). In this state, a write by one of the sharers sends the update, rather than the cache line, through the wireless network, and updates the caches of all the other sharers. A read by a sharer gets the latest version of the line from its cache. The wireless network serializes all the updates to any lines in W state. In W state, the directory does not record which cores share the line, but only how many do. A line enters the W state from the Shared (S) state of the wired protocol, when the number of sharers goes above a *MaxWiredSharers* threshold. Cores with a line in W state are supposed to actively share the line, by regularly reading/writing the line. If a core does not do that, the hardware sends a signal to the directory, which decreases the count of wireless sharers. When the count decreases to *MaxWiredSharers*, the line transitions to the S state of the wired protocol.

The directory structure changes little from a conventional design. Without loss of generality, *WiDir* builds on top of a conventional MESI protocol, with a directory implementation that uses i shared pointers with broadcast (Dir_iB) [7], [8]. However, many other implementations, such as a Coarse Vector design (Dir_iCV_r) [8] can easily be adapted as well. The one constraint is that *MaxWiredSharers* is no higher than the number of sharer pointers supported by the directory scheme — i.e. i in Dir_iB or Dir_iCV_r.

Figure 3 shows the structure of the directory and cache in the conventional protocol augmented with *WiDir*. The figure also shows the changes added by *WiDir* in bold. First, one of the stable states is W. Second, when a directory entry changes to W, the sharer pointer field becomes a *count of the sharer cores* (*SharerCount*), and the Broadcast bit becomes the *Wireless* bit, which is set. Note that the field with the count of sharer cores needs to have as many as $\log_2 N$ bits, where N is the core count in the manycore. This is because a wireless line may be shared by all the cores in the manycore.

To understand how *WiDir* works, we describe the two main transactions, namely the S→W transition and the W→S transition. In the process, we will see the need for two new primitives for wireless cache-coherence protocols, namely the *Selective Wireless Data-Channel Jamming* (*Jamming*) and the *Wireless Tone Channel Acknowledgment* (*ToneAck*). The former gives the directory the ability to reject incoming transactions on a directory entry that the directory is currently operating on; the corresponding primitive in *wired* protocols is bouncing (or buffering) incoming transactions if the directory entry is busy. The second primitive gives the directory the ability to receive acknowledgment messages from many cores very cheaply. The primitives are discussed in Section III-C.

1) Transition from Shared to Wireless:

When the directory receives a read/write request from a new core for a line that is already shared by *MaxWiredSharers* cores, the directory initiates the transition of the line to W. It does so by broadcasting a BroadcastWirelessUpgrade (*BrWirUpgr*) message for the line on the wireless data channel. It also sends a WirelessUpgrade (*WirUpgr*) response with the line to the requesting cache using the wired network.

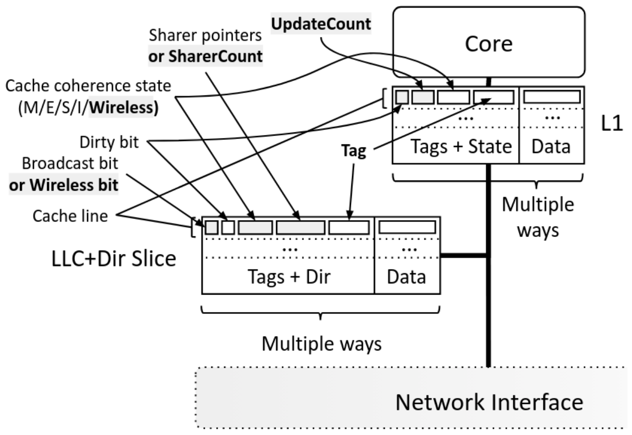


Fig. 3: Cache and directory structure of conventional protocol augmented (in bold) for *WiDir*.

Immediately on reception of *BrWirUpgr*, all the Tone antennas in the manycore (including the one in the node with the directory that initiated the message) initiate a *ToneAck* operation. For the initiating directory, this operation fully terminates only when each of the nodes completes the following: (i) the node determines that it does not contain the line in its private cache, (ii) the node finds out that it contains the line and sets its cache state for the line to W, (iii) the node had requested the line using the wired network and has finally received either the line or a bounced response from the directory; if it has received the line, it has set its cache state to W.

Once the *ToneAck* operation is complete, the directory stores a count of the sharer nodes (*SharerCount*) in the sharer pointers field, changes the state to W, and sets the Wireless bit. From now on, all writes by the sharer processors use the wireless data network. In particular, the core that, by issuing a request triggered the transition to W, if it intended to write, it now retries the write using the wireless data network.

However, nodes complete their *ToneAck* operation at different times. When they do, they resume execution. It is possible that one of the sharer nodes now issues a write (now using the wireless network) before the directory has fully terminated the *ToneAck* operation. The directory has to prevent this from happening because its transition is not completed. Consequently, after initiating the *ToneAck* operation, the antenna in the node with the home directory initiates a *Jamming* operation for this line in the data channel. This prevents any core from successfully updating this line using the wireless data channel.

Finally, if a new core issues a read/write request to the line, the transaction will reach the directory using the wired network. In this case, the directory responds to the requester with *WirUpgr* and the line, using the wired network and, importantly, increments *SharerCount*.

2) Transition from Wireless to Shared: When cores keeping a line in W state are not interested in frequently sharing the line anymore, the line returns to S. To be able to do so, *WiDir* augments each line in the private cache with a short counter (e.g., 2 bits) called *UpdateCount* that detects when the local core is not interested in the line anymore. When a cached line enters the W state, *UpdateCount* is cleared. From then on, every time that the cache receives a wireless update, *UpdateCount* is incremented; every time that the local core accesses the line, *UpdateCount* is reset. If *UpdateCount* reaches a certain threshold, it is assumed that the local core is uninterested in the line. Hence, the hardware evicts the line from the cache and sends a message PutWireless (*PutW*) to the directory indicating that the core is no longer a sharer. *PutW* is sent through the wired network to avoid consuming wireless bandwidth for such a non-critical message.

When the home directory receives a *PutW* for a line, it decrements the *SharerCount* for the line. If the counter reaches *MaxWiredSharers*, the line should transition to S.

To do so, the directory broadcasts a WirelessDowngrade (*WirDwgr*) for the line on the wireless data channel. As each node receives the message, its cache controller checks if the cache indeed has the line. If it does not, no action is taken, otherwise, an acknowledgment message is sent to the directory with the sender's node ID. These messages use the wired network to save wireless bandwidth. When the directory receives all the *MaxWiredSharers* acknowledgments expected, it stores the sharer IDs in the Sharer Pointer field in the directory entry and clears the Wireless bit. In addition, if the LLC line is Dirty, it is written to memory. Finally, the state is set to S. The transaction is now complete and the directory accepts new requests. From now on, all communication occurs via the wired network.

When a core evicts a W line from its private cache, it also sends a *PutW* through the wired network to the directory, which will decrease *SharerCount*. In addition, to keep the directory up to date, a node always informs the directory when any line is evicted from its private cache. While this is not strictly needed for non-W lines to attain the functionality we desire, we do it for simplicity.

C. Primitives for Wireless Protocols

To support efficient wireless cache coherence protocols, we propose the following two primitives.

1) Selective Wireless Data-Channel Jamming: Conventional cache coherence protocols provide support for the directory to stop (i.e., buffer) or reject (i.e., bounce) new transactions directed to a directory entry that is currently busy. We propose to provide a similar primitive, called *Jamming*, for wireless directory protocols. Jamming builds on the Broadcast, Reliability, Sensing (BRS) wireless protocol [37]. In that protocol, a transceiver eagerly starts sending a message as soon as it appears there is a free cycle in the wireless network. In the first cycle, the message includes most of the address. Then, in the second cycle, the transceiver stops transmitting and, instead, listens to whether any other transceiver reports (with a brief negative-Ack) a collision in the first cycle. If there is indeed a reported collision, the message is retried (e.g., after an exponential back-off period). Otherwise, in the third and subsequent cycles, the rest of the message is sent out. No further collision is possible because no transmitter will send any other message until the current one uses up its allocated wireless cycles.

With this support, when a directory temporarily wants to prevent any new wireless transaction on a line, it proceeds as follows. It directs its transceiver to listen to every message initiation in the wireless data-channel network. If the first cycle of the message includes an address equal to the line's address (or potentially equal if all the bits were available), the transceiver then forces an interruption of the message by sending a negative-Ack, similarly as if a collision occurred. As a result, the message will be aborted. With this support, the directory prevents transactions to the line (with some false positives), but enables transactions to other lines.

2) Wireless Tone Channel Acknowledgment: In conventional cache coherence protocols, some transactions require the collection of acknowledgement messages in the directory. Such messages take a long time to complete and, in addition, cause network contention. We propose to support a primitive for wireless directory protocols called *ToneAck* that allows the directory to receive acknowledgments from many processors very cheaply. In fact, since wireless messages are broadcasted, ToneAck involves an acknowledgment from all cores.

ToneAck is triggered when an initiating transceiver issues a certain wireless event — e.g., the node of the home directory issues a *WirUpgr* message. ToneAck consists of all the transceivers producing a continuous tone in the Tone channel, and the initiating transceiver monitoring the tone. In parallel, each node performs a certain operation (which may be a simple check to determine that no action is needed) and, once completed, removes its tone from the Tone channel. Once the initiating transceiver notices that there is no tone in the channel, it knows that all nodes have completed their task.

Effectively, ToneAck enables a fast global acknowledgment operation. Similar support has been proposed by TLSync [38] and WiSync [19] for efficient core synchronization. However, this is the first time that this idea is used as part of the transactions of a cache coherence protocol.

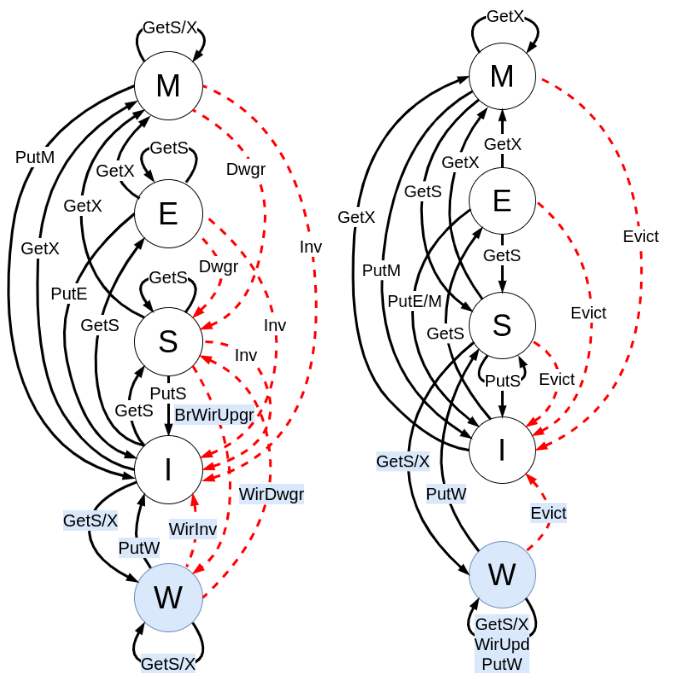
D. Summary: Why Adding Wireless Support To Coherence

Adding support for wireless communication in a large manycore provides the ability to perform several types of coherence transactions very efficiently, especially those involving update multicast. While the bandwidth of a wireless network is limited, the latency of any given transaction is very small. These properties perfectly fit the sharing pattern considered in this paper: groups of cores frequently reading and writing a shared location. Now, read and write operations do not need the complicated multi-hop protocol transactions required by invalidation-based protocols, or the lengthy routing of messages required by invalidation- or update-based protocols in meshes and other wired networks. Instead, writes transfer a single update — rather than a cache line — with about 5ns [19], and reads are local.

IV. Detailed Design

A. Protocol Description

Figures 4a and 4b show the diagram of all possible transitions between stable states of *WiDir* in the controller of the private caches, and directory, respectively. The transitions are annotated with descriptive labels. As seen in the figures, we have the four MESI states plus the wireless (W) state.



(a) L1 cache controller. (b) Directory controller.

Fig. 4: Transitions between stable states of *WiDir* in the controllers of the private caches (a), and directories (b). In the figures, black lines are core-initiated transactions; red dashed lines are directory-initiated transactions; and the blue text is the added wireless coherence transitions.

Rather than describing the complete protocol in detail, we focus only on the transitions that come from W or go to W. Tables I and II describe such transitions for the controllers of the private caches and directories, respectively. Each row considers one transition and describes when the transition happens and the action taken.

Private Cache Controller (Table I). There are four cases of I→W transitions. The first two are when the directory is in W and the core issues a GetS or GetX (indicating that the core is not a sharer) to the directory. In this case, the cache receives, via the wired network, a wireless upgrade message (*WirUpgr*) plus the line from the directory, and sends back a wireless upgrade acknowledgment (*WirUpgrAck*) to the directory via the wired network. The cache line transitions to W. If the request was a GetX, the cache now issues the update wirelessly.

The other two cases are when the core issues a GetS or GetX (again, indicating that the core is not a sharer) to the directory that triggers a directory transition to W. In this case, the local transceiver receives a broadcast wireless upgrade message (*BrWirUpgr*) via the wireless network, and turns on the tone channel. Then, when the cache receives, via the wired network, a *WirUpgr* plus the line from the directory, it transitions to W and tells the transceiver to turn off the tone channel. If the request was a GetX, the cache issues the update wirelessly.

There are two cases of S→W transitions. The first one is when the transceiver receives a *BrWirUpgr* message via wireless from the directory because the latter transitions to W. In this case, the transceiver turns on the tone channel, the cache transitions to W, and the transceiver turns off the tone channel.

The second case is when the core issues a GetX to the directory (indicating that it is already a sharer) and, by the time it gets to the directory, the latter has transitioned to W. In this case, the transceiver receives a *BrWirUpgr* via wireless, and turns on the tone channel. The cache transitions to W and the transceiver turns off the tone channel. Finally, the cache issues the update wirelessly.

There are two cases of W→W transitions. The first one is when the core reads; in this case, the hardware reads from the cache and clears the *UpdateCount* for the line. The second case is when the core writes. In this case, the transceiver broadcasts the updated word (*WirUpd*) via the wireless data network. When the transceiver indicates that the broadcast has succeeded, the local cache is also updated. The *UpdateCount* for the line in the cache is cleared.

There is one case W→S transition. It is when the cache receives a wireless downgrade message (*WirDwgr*) from the directory via wireless because *SharerCount* decreased to *MaxWiredSharers*. In this case, the cache controller sends a wireless downgrade acknowledgment message (*WirDwgrAck*) that includes the core ID to the directory via the wired network. The line state is changed to S.

There are two cases of W→I transitions. The first is when the local cache evicts a line in W state. In this case, the cache controller informs the directory by sending it a *PutW* message via the wired network. The second case is when the local cache receives a wireless invalidate message (*WirInv*) for line from the directory via wireless because the directory is evicting the line. The cache invalidates the line and, if the core has a pending write on the line, it squashes it and retries it.

Directory Controller (Table II). The transition S→W occurs when the directory receives a GetS or GetX from a non-sharer node via the wired network, and the new number of sharers for the line is now higher than MaxWiredSharers. In this case, the directory broadcasts a *BrWirUpgr* via the wireless network, turns on jamming in the wireless data network for the line and, via the wired network, sends *WirUpgr* plus the line to the requester node. Once the directory detects that the tone channel is silent, it sets the Wireless bit, sets *SharerCount* to the count of sharers, sets the state to W, and turns off jamming.

TABLE I: State transitions that come from W or go to W for the controller of the private cache.

Trans.	When?	Action
$I \rightarrow W$	Core issues GetS to directory and directory is in W	Cache receives a *WirUpgr*+line via wired, sends *WirUpgrAck* back to directory via wired, and transitions to W
	Core issues GetX to directory (indicating that it is not a sharer) and directory is in W	Cache receives a *WirUpgr*+line via wired, sends *WirUpgrAck* back to directory via wired, and transitions to W. The cache issues the update wirelessly
	Core issues GetS to directory triggering a directory transition to W	Transceiver receives a *BrWirUpgr* via wireless, and sets the tone channel on. When the cache receives a *WirUpgr*+line via wired, it transitions to W and notifies the transceiver to turn off the tone channel
	Core issues GetX to directory (indicating that it is not a sharer) triggering a directory transition to W	Transceiver receives a *BrWirUpgr* via wireless, and sets the tone channel on. When the cache receives a *WirUpgr*+line via wired, it transitions to W and notifies the transceiver to turn off the tone channel. The cache issues the update wirelessly
$S \rightarrow W$	Transceiver receives a *BrWirUpgr* from the directory via wireless because the latter transitions to W	Transceiver turns on the tone channel, the cache transitions to W, and the transceiver turns off tone channel
	Core issues GetX to directory (indicating that it is already a sharer) and, by the time it gets to the directory, the latter has transitioned to W	Transceiver receives a *BrWirUpgr* via wireless, and turns on the tone channel. The cache transitions to W and the transceiver turns off the tone channel. Finally, the cache issues the update wirelessly
$W \rightarrow W$	Core reads	Clear *UpdateCount*
	Core writes	Transceiver broadcasts the updated word (*WirUpd*) via wireless. When the transceiver indicates that the broadcast succeeded, the local cache is updated and *UpdateCount* is cleared
$W \rightarrow S$	Cache receives a *WirDwgr* from the directory via wireless because *SharerCount* decreased to *MaxWiredSharers*	Cache sends *WirDwgrAck* (including core ID) to directory via wired, and changes the state to S
$W \rightarrow I$	Local cache evicts W line	Cache notifies the directory with *PutW* via wired
	Cache receives a *WirInv* via wireless from the directory because the directory wants to evict the line	Cache invalidates the line and, if the core has a pending write on the line, it squashes it and retries it

TABLE II: State transitions that come from W or go to W for the directory controller.

Trans.	When?	Action
$S \rightarrow W$	Directory receives a GetS or GetX from a non-sharer cache via wired, and new number of sharers is higher than *MaxWiredSharers*	Directory broadcasts *BrWirUpgr* via wireless, turns on jamming on the line, and sends *WirUpgr*+line to the requester via wired. Once the tone channel is silent, directory sets the Wireless bit, sets *SharerCount*, sets state to W, and turns off jamming
$W \rightarrow W$	Directory receives from a cache via wired a GetS or GetX (indicating that is not a sharer)	Directory turns on jamming, and sends *WirUpgr*+line via wired to the requester. Upon receiving its *WirUpgrAck* via wired, increase *SharerCount* and turns off jamming
	Directory receives a GetX from a cache via wired (indicating that it already is sharer) not knowing the directory is in W	Directory discards message since a previously sent *BrWirUpgr* via wireless already resolved the conflict.
$W \rightarrow S$	Directory receives a *PutW* via wired. The *SharerCount* is decreased and now becomes *MaxWiredSharers*	Directory broadcasts *WirDwgr* wirelessly and waits for *WirDwgrAck* acknowledgments via wired. Upon receiving all *MaxWiredSharers WirDwgrAck*s, the directory records their core IDs in the sharer pointers, resets the Wireless bit, writes the line to memory if it is dirty in the LLC, and sets the state to S
$W \rightarrow I$	Directory evicts a line shared wirelessly	Directory broadcasts a *WirInv* via wireless. If the line is dirty in the LLC, the line is written to memory

The transition W→W occurs in two cases. The first one is when the directory receives from a cache via the wired network a GetS or GetX (indicating that the node not a sharer). In this case, the directory turns on jamming and sends, via the wired network, a *WirUpgr* plus the line to the requester. When the directory receives a *WirUpgrAck* via the wired network, it increases *SharerCount* and turns off jamming.

The second case is when the directory receives a GetX from a cache via the wired network indicating that the node is already a sharer, but not knowing that the directory is already in W. In this case, the directory discards the message since a previously-sent *BrWirUpgr* via wireless already resolved the conflict.

The W→S transition occurs when the directory receives a *PutW* message via the wired network, as a result of a wireless sharer evicting the line from its cache. After the directory decrements *SharerCount*, the latter becomes *MaxWiredSharers*. In this case, the directory broadcasts *WirDwgr* wirelessly, and waits for *WirDwgrAck* acknowledgments from *MaxWiredSharers* cores via the wired network. Upon receiving them, the directory records their core IDs in the sharer pointers, resets the Wireless bit, writes the line back to memory if the Dirty bit in the LLC is set, and sets the state to S.

The transition W→I occurs when the directory evicts a line shared wirelessly. In this case, the directory broadcasts a *WirInv* message via the wireless network. If the Dirty bit in the LLC is set, the line is written to memory.

B. Correct Operation of the Wireless Update Transactions

Writes that use the wireless network, like all other writes, are kept in a core's write buffer until they complete. The process of performing such a write involves multiple steps. The first one is for the transceiver to obtain access to the wireless data network. The transceiver then has to succeed in sending the message without collisions. Once the message is fully transmitted (i.e., 5 cycles from the initiation), the transceiver signals the private cache controller to merge the write into the local cache. Once the merging is done, the write is complete.

Before the transceiver gains access to the wireless network, the transceiver may receive updates to the line from remote cores. In this case, such updates are performed before the local one. It is also possible that the transceiver receives an invalidation for the line (*WirInv*) because the line's entry in the directory was evicted. In this case, the local cache line is invalidated and the local write is squashed and retried. As the local write retries, it will trigger the directory to allocate a new entry for the line.

The fact that the update has been successfully transmitted via the wireless data network does not imply that the update has already updated the home directory or the remote caches. This is because, in each destination node, the update has to be routed to the controllers for the private cache and (in the home node) to the directory/LLC controller (Figure 2). Such routing takes some delay. For the coherence protocol to be correct, we need to ensure that the update is not overtaken by requests that are issued later. To ensure that this does not happen, *WiDir* has a single queue for the private cache controller, and a single queue for the directory/LLC controller. Further, messages are processed in strict FIFO order. Specifically, messages to the private cache controller from the core, transceiver, router, and directory/LLC in Figure 2 all go to a single FIFO queue. The same applies to the directory/LLC controller. Hence, as an incoming update is received by the transceiver, it gets ordered in the queue where it goes to.

V. Simulation Methodology

We use the SST [39] simulation framework as a cycle-accurate, execution-driven simulator to model a server architecture with 64 cores. The architecture parameters, as well as the energy and area of the wireless components are shown in Table III. Each processor tile is composed of an out-of-order core with private L1 instruction and data caches. L2 is shared and physically distributed across the processor tiles. For the NoC, we use a 2D-mesh topology with a latency of 1 cycle per hop. We augment the simulator to model in detail the transmissions and collision handling required by the wireless network. The data channel of the wireless network has a bandwidth of 20 Gb/s. This is adequate to transmit a word and its address, required for broadcasting data messages and the smaller sized coherence messages, within 4 cycles. Collision detection requires one additional cycle.

We use McPAT [45] and CACTI [46] to model the energy consumed by cores and memory hierarchy, as well as a calibrated DSENT [47] to model the energy of the wired links and routers. To compute the power and area consumed for the wireless hardware (transceiver, data converter, serializer, deserializer, and antenna) we use and adapt published data in 65nm technology [18], [40]–[44], [48]. Because some of the components' data were given only for 16 Gb/s, we linearly scaled up their power and area as needed, to match our 20 Gb/s bit rate. Note that the power and area of the wireless components would be lower if we scaled them to more recent

TABLE III: Architecture modeled. RT means round trip.

General Parameters	
Architecture	Manycore with 64 cores
Core	Out of order, 4-issue wide, 1GHz, x86 ISA
ROB; ld/st queue	64 entries; 20 entries
L1 I+D caches	Private 64KB WB, 2-way, 2-cycle RT, 64B lines
L2 cache	Shared, 512KB WB banks per-core (32 MB total)
L2 bank	8-way, 12-cycle RT (local), 64B lines
On-chip network	2D-mesh, 1 cycle/hop, 128-bit links
Off-chip memory	Connected to 4 memory controllers, 80-cycle RT
WiDir Parameters	
Cache Coherence	*WiDir*-MESI directory based
Data Wireless channel	20Gb/s; 5-cyc. transfer lat. (1-cycle collision detection)
Tone Wireless channel	1Gb/s; 1-cycle transfer latency
MaxWiredSharers	3 sharers/line
MAC Protocol	BRS (exponential backoff)
Transceiver	0.25mm^2, 30mW [18], [40], [41]
Data converter	0.03mm^2, 0.72mW [42]
Serializer/deserializer	0.04mm^2, 10.8mW [43]
Data+tone antennae	0.08mm^2 [44]
Overall RF circuit	Area: 0.4mm^2; Tx/Rx/idle: 39.4/39.4/26.9mW (power gating analog amplifier, with transient energy of 1.14 pJ)

CMOS technology, as proposed by other researchers [49], [50]. However, in this paper we choose to be conservative and not scale them down. Finally, the energy parameters of the wireless components in Table III also take into consideration the fact that the transmitter's power amplifier and the receiver's low noise amplifier can be power gated when not in use [18], [51].

To evaluate the efficacy of our design, we compare it against a *Baseline* system with no wireless support for a wide range of multi-threaded applications. These applications, which are listed and characterized by their L1 misses-per-kilo-instructions (MPKI) in Table IV, show different levels of fine-grained data sharing, as well as different access patterns.

For example, as shown by Barrow-Williams *et al.* [52], *FFT* has broadcast communication; *Blackscholes* contains mostly communication between neighbors; *Canneal* has an irregular communication pattern, due to locks; and *Bodytrack* has one-to-many communications and reductions. As a result, we evaluate *WiDir* across different application characteristics.

For the SPLASH-3 applications, we use the default input sets as described in [53], and, for PARSEC, we use the standard *simsmall* [54]. The input set sizes were chosen to allow running each benchmark to completion within a realistic time frame of up to a few days of simulation.

TABLE IV: Simulated applications characterized by L1 MPKI

SPLASH-3 [53]				**PARSEC** [54]	
Name	**MPKI**	**Name**	**MPKI**	**Name**	**MPKI**
Water-spa	0.49	*Water-nsq*	2.86	*Blackscholes*	0.13
Barnes	9.53	*Radix*	9.41	*Bodytrack*	7.51
Ocean-nc	16.05	*Ocean-c*	16.14	*Canneal*	23.21
FMM	1.88	*Cholesky*	5.92	*Dedup*	4.1
Volrend	2.44	*Radiosity*	5.28	*Fluidanimate*	1.27
Raytrace	10.05	*FFT*	5.05	*Freqmine*	8.84
LU-nc	21.52	*LU-c*	1.9	*Ferret*	6.34

VI. Results

We analyze data sharing, then evaluate performance, energy, and area, and finally perform a sensitivity analysis.

A. Data Sharing Analysis

Figure 5 considers all the wireless updates in *WiDir*, and shows a histogram of the number of sharers that are updated per write. The figure shows that, for many applications, a wireless write updates many caches. For other applications, only a few caches are typically updated. On average, we see that updates with few sharers (up to 5) account for 36% of all writes. On the other hand, updates with many sharers (50+) account for 37% of the writes. The latter correspond to highly shared variables, such as locks and barriers. This category is the one that offers the highest benefit from the wireless mode.

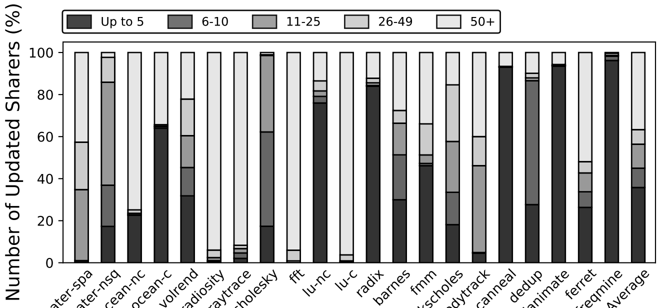


Fig. 5: Number of wireless sharers updated upon a write.

Note that the detection of lines with many sharers does not require any user effort. The hardware automatically identifies them and switches them to wireless mode. There is also no need for any dedicated wireless memory.

B. Performance Analysis

Cache Misses. Figure 6 shows the L1 misses per kilo instruction (MPKI) of *WiDir* normalized to the baseline configuration. The figure is broken down into read and write misses. We can see that, on average, the number of misses is reduced by 16%. The misses eliminated are coherence misses. *WiDir* removes them by virtue of updating the sharers of a wireless line upon a write, as opposed to invalidating them.

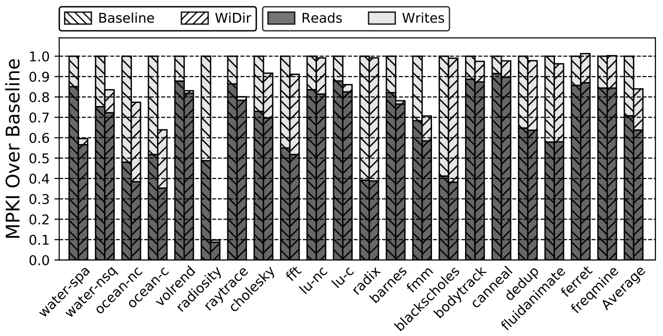


Fig. 6: Misses-per-kilo-instruction in *WiDir* over baseline.

The reduction in MPKI is especially large in *radiosity*. For this application, Figure 5 showed that 96% of the wireless writes in *WiDir* update 50+ sharers. Updating so many sharers, as opposed to invalidating them, achieves a large reduction in MPKI. Other applications that see a large reduction in MPKI are *water-spa, ocean-nc, ocean-c, raytrace, barnes*, and *fmm*. Most of these applications have a large average number of wireless sharers updated per write (Figure 5). However, the two figures are not perfectly correlated because there are also misses to non-wireless lines.

Memory Latency. Figure 7 shows the average latency of memory operations in *WiDir*, normalized to that of the baseline. This latency is counted as the number of cycles from the time each request enters the ROB, until it retires. The figure is broken down into reads and writes.

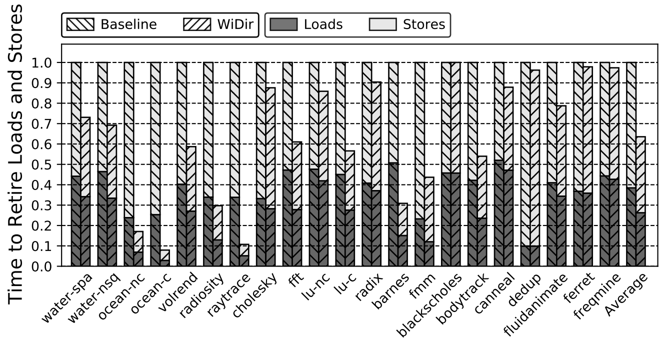


Fig. 7: Latency of memory operations in *WiDir* over baseline.

We see that the reduction in average memory access latency of *WiDir* is large for many applications. On average, the latency reduction of memory operations in *WiDir* over the baseline is 36%. Applications such as *ocean-nc, ocean-c, radiosity, raytrace, barnes*, and *fmm* have large latency reductions. Typically, this is because these applications experienced a substantial reduction in L1 MPKI in Figure 6. On the other hand, applications that did not see reductions in L1 MPKI in Figure 6, such as *radix, blackscholes, dedup, ferret*, and *freqmine*, do not show reductions in average memory access latency either. Again, the correlation is not perfect because the change in the average memory access latency also depends on the original relative weight of misses and hits.

Wired Network Hops. In a conventional wired protocol, read and write miss transactions often require multiple coherence hops (or *legs*), as the directory forwards messages to other sharers. In a large chip, such as the 64-core manycore that we are simulating, each of these legs of a coherence transaction ends up incurring a high cost due to the large number of network hops each message has to go through in the wired mesh.

To assess this cost, we count, for each leg of a coherence transaction, the number of network hops that the message has to go through. Table V shows the average number of such network hops per transaction leg in the wired mesh of our 64-core baseline architecture. We can see that more than half of all the messages in the baseline have to perform at least 6 network hops to reach their destination. In contrast, in *WiDir*, writes to wireless lines are broadcasted to all sharers in a single hop. This reduces the latency.

TABLE V: Average number of network hops for all messages sent through the wired mesh, in the 64-core baseline architecture.

Number of Hops	***0-2***	***3-5***	***6-8***	***9-11***	***12-16***
% of Messages	16%	23%	32%	21%	8%

Execution Time. Figure 8 presents the execution time of the applications in *WiDir* normalized to the baseline, for 64 cores. Each bar is broken down into three different types of execution cycles: cycles where the execution is stalled due to a pending memory access, cycles where the execution is stalled due to other reasons (e.g., thread synchronization or load imbalance), and cycles where instructions are being committed.

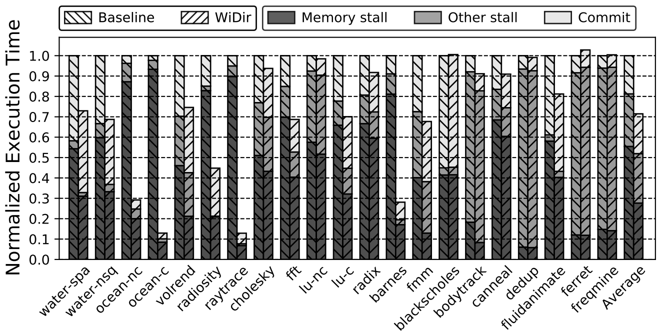


Fig. 8: Normalized execution time of *WiDir* to baseline, broken down into the different types of cycles.

We can see that, in the baseline environment, many of the cycles are wasted to memory stall. On average, more than half of the cycles fall into this category. With *WiDir*, there are large reductions in the execution times of many applications. Such reductions are mostly caused by decreases in the memory stall cycles. On average, *WiDir* manages to reduce about half of the memory stall time. As a result, on average, the total execution time of the applications reduces by 28%. With *WiDir*, each of the types of cycles accounts for about a third of the execution time on average.

Generally, the applications with the highest percentage of memory stall cycles are the ones that benefit the most from *WiDir*. They include *ocean-nc, ocean-c, radiosity, raytrace*, and *barnes*. This is because wireless transactions directly try to reduce the number of misses and, hence, the memory stall time. In applications where the stall comes from other reasons, such as *bodytrack, dedup, ferret*, and *freqmine*, the impact of *WiDir* is minor.

C. Energy and Area Analysis

Energy. Figure 9 shows the energy consumed by *WiDir* relative to the baseline. The energy is broken down into the energy of the core, the private L1, the shared L2+directory, the wired NoC, and the wireless transceiver.

We can see that the baseline spends on average about 60% of the energy in the core, 5% in the instruction and data L1 caches, 20% in the shared L2, and 15% in the wired NoC. The energy savings of *WiDir* over the baseline are on average 29%,

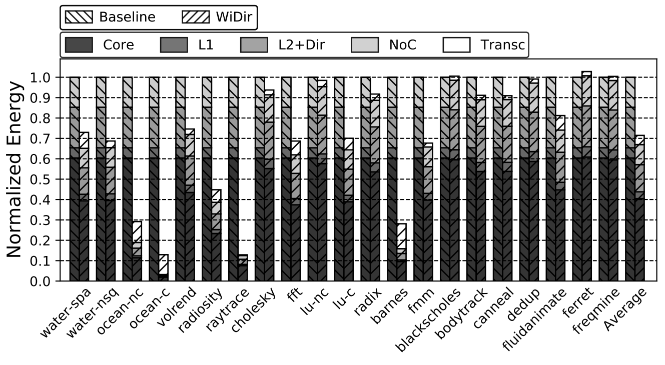


Fig. 9: Energy consumed by *WiDir* relative to baseline.

and mostly proportional to the performance improvements in each of the applications. The wireless network and coherence protocol reduce the cost of polling operations and long distance communications. Therefore, they reduce the energy consumption.

The energy cost of the wireless communication can be estimated by the energy contribution of the transceiver. This contribution is modest: on average, it is 6.29% of *WiDir* energy.

Finally, since the energy reduction of *WiDir* is roughly similar to the execution time reduction (Figure 8), we conclude that the power consumed by baseline and *WiDir* are very similar.

Area. Based on the numbers presented in Table III, we estimate that the area overhead of *WiDir* over a system without wireless support amounts to 4.05%. This is a very small number. It includes the contributions of the transceivers, data converters, serializers, deserializers, and antennas. Note that these results are conservative, as we do not scale the area of these components from 65 nm (Section V). The 4.05% area overhead also includes the *UpdateCount* bits, but the area of such bits is negligible.

D. Speedup and Sensitivity Analysis

As the number of cores in the manycore increases, so does the overhead of traditional coherence protocols, as well as the cost of traversing the wired mesh. While this is also true for *WiDir*, the added wireless coherence protocol reduces the overhead of the lines that are highly shared and would suffer the most. The result is a better scalability of *WiDir* with the number of cores than the baseline.

Figure 10 shows the execution speedup of *WiDir* and the baseline as the number of cores increases. The speedups are computed relative to the execution time of the baseline with four cores. From the figure, we can see that, for up to 16 cores, the difference between the execution time of *WiDir* and the baseline is small. This is because the cost of traversing the wired network is small, and the number of cores sharing a cache line is typically modest. As a result, *WiDir* cannot provide much benefit. However, as the size of the chip increases to 32 and 64 cores, *WiDir* benefits from having more cache lines in wireless mode, while the baseline is harmed by the increased cost of traversing the wired network. Thus, the average speedup for *WiDir* keeps increasing, while the baseline saturates. Hence, *WiDir* is more scalable.

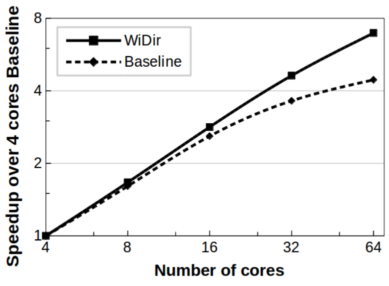


Fig. 10: Average execution time speedup of *WiDir* and baseline over the 4-core baseline.

Max Wired Sharers	Sp.	Coll. prob.
2	*1.94x*	*6.64%*
3	*2.10x*	*2.76%*
4	*2.02x*	*2.61%*
5	*1.98x*	*2.04%*

TABLE VI: Speedups of *WiDir* over baseline, and collision probability in *WiDir*, for different values of *MaxWiredSharers*.

We now consider the effect of the threshold for the number of cores that need to be sharing a line for the line to switch to wireless (*MaxWiredSharers*). Recall that the default value of *MaxWiredSharers* is three. In this section, we change its value to 2, 4, and 5.

For the different values of *MaxWiredSharers*, Table VI shows two measures: (i) the average execution time speedup of *WiDir* over the baseline, for 64-core runs (*Sp.*), and (ii) the probability of collisions of messages in the wireless network in *WiDir* (*Coll. prob.*).

When we switch to wireless mode sooner (i.e., *MaxWiredSharers=2*), more lines are in wireless mode. This increases contention and collisions in the wireless medium (from 2.76% to 6.64%), and in turn hurts speedups (from 2.10x to 1.94x) when compared to the default *MaxWiredSharers* value of three. It can be shown, however, that some applications such as *ocean-nc, lu-nc, fmm*, and *canneal*, actually attain higher speedups when switching to wireless mode sooner.

When we increase the threshold to switch to wireless later (*MaxWiredSharers*=[4, 5]), fewer lines are in wireless mode. Hence, we reduce the amount of traffic in the wireless medium, which in turn decreases the probability of wireless collisions (from 2.76% to 2.61% and 2.04%). However, the result is reduced speedups (from 2.10x to 2.02x and 1.98x), due to missing opportunities to use the wireless medium. No application attains higher speedup with these higher *MaxWiredSharers* values.

Overall, our default *MaxWiredSharers* value works best.

VII. Related Work

Wireless Architectures. We described WiSync [19], Choi *et al.* [20], and Replica [21] in Section II. Other works use a WNoC to accelerate the communication patterns of applications such as graph analytics [55], molecular dynamics simulations [56], and brain-machine interfaces [57]. Those works differ from *WiDir* in that the wireless links are used to reduce the average network latency regardless of the coherence state. Further, the network is optimized for a particular set of applications only.

Broadcast-based Protocols. *WiDir*'s wireless network provides a totally-ordered interconnect and efficient broadcasting to satisfy misses with low latency and resolve protocol races. Snooping protocols [58] also typically use a totally-ordered interconnect to those same ends. Several works extend snoopy coherence to unordered interconnects to improve its scalability [59]–[65]. The main idea is to provide ordering guarantees, either at the protocol level via tokens associated to each cache line [60], or at the network level using global timestamps [59], snoop-order numbers [62], a logical embedded-ring [61], or a dedicated ordering network [63]–[65]. Depending on the protocol, however, the scalability is fundamentally limited by the increasing ordering buffer requirements, latency of serialization, and inefficiency of filling a mesh NoC with broadcasts. *WiDir*, instead, avoids filling the mesh and is naturally scalable.

Protocols with Emerging Interconnect Technologies. Optical/RF transmissions via shared nanophotonic waveguides [66]–[70] or transmission lines (TLs) [38], [71]–[75] can provide ordered broadcast. Some proposals use these global interconnects to implement conventional snoopy coherence [66], custom limited directory protocols [68], race-free protocols via globally shared locks [67], or to speed-up synchronization [38], [76]. Compared to wireless networks, both nanophotonics and TLs are more energy efficient and provide higher bandwidth because of their guided nature. However, network design becomes more complex than wireless because a maze of physical waveguides/TLs needs be planned and laid down. Both technologies also have scalability issues in globally shared networks. In nanophotonics, it is because of the losses added by each and every power divider, modulator, coupler, and receiver along the path. In TLs, it is because of the need for a centralized arbiter for the bus, and of overcoming signal reflections with amplifying stages between segments, which are costly and complicate the design.

Scalable Directory Protocols. We outlined a variety of protocols with scalable directories [7], [8], [34] in Section II. Another way to scale the size of the directory is by dividing the manycore in coherence domains [77], where only the cores in the same domain are kept coherent. *WiDir*, instead, not only allows the directory to scale without domain restrictions, but also boosts performance by enabling instantaneous updates of highly-shared lines.

VIII. Conclusion

To handle sharing patterns where a group of cores frequently reads and writes a set of shared variables, this paper used on-chip wireless network technology to augment a conventional directory-based invalidation cache coherence protocol. The resulting protocol, called *WiDir*, seamlessly transitions between wired and wireless coherence transactions for the same data based on the program's access patterns in a programmer-transparent manner. In this paper, we described the protocol in detail. Further, an evaluation showed that *WiDir* substantially reduces the memory stall time of applications. For 64-core runs, applications took on average 28% less time to complete on *WiDir* than on MESI. Moreover, *WiDir* was shown to be more scalable than MESI. These benefits were obtained with modest area and power costs.

References

[1] *Ampere Altra 64-Bit Multi-Core ARM Processor*, 2020 (accessed April 16, 2020).

[2] *AMD Epyc 7742 Processor*, 2019 (accessed April 16, 2020).

[3] *Intel Xeon Platinum 9282 Processor*, 2019 (accessed April 16, 2020).

[4] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1997.

[5] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-gneration Intel Xeon Phi product," *IEEE Micro*, 2016.

[6] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor," *IEEE Computer*, 1992.

[7] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1988.

[8] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1990.

[9] H. Grahn, P. Stenstrom, and M. Dubois, "Implementation and evaluation of update-based cache protocols under relaxed memory consistency models," in *Future Generation Computer Systems*, 1995.

[10] S. Abadal, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio, "OrthoNoC: A broadcast-oriented dual-plane wireless network-on-chip architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 628–641, 2017.

[11] G. Ascia, V. Catania, S. Monteleone, M. Palesi, D. Patti, J. Jose, and V. M. Salerno, "Exploiting data resilience in wireless Network-on-Chip architectures," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 2, pp. 1–27, 2020.

[12] D. DiTomaso, A. Kodi, D. Matolak, S. Kaya, S. Laha, and W. Rayess, "A-WiNoC: Adaptive wireless Network-on-Chip architecture for chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3289–3302, 2015.

[13] A. Karkar, T. Mak, N. Dahir, R. Al-Dujaily, K.-F. Tong, and A. Yakovlev, "Network-on-Chip multicast architectures using hybrid wire and surface-wave interconnects," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 3, pp. 357–369, 2018.

[14] K. Duraisamy, Y. Xue, P. Bogdan, and P. P. Pande, "Multicast-aware high-performance wireless Network-on-Chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 3, pp. 1126–1139, 2017.

[15] D. W. Matolak, A. Kodi, S. Kaya, D. Ditomaso, S. Laha, and W. Rayess, "Wireless networks-on-chips: architecture, wireless channel, and devices," *IEEE Wireless Communications*, vol. 19, no. 5, pp. 58–65, 2012.

[16] S. H. Gade and S. Deb, "HyWin: Hybrid wireless NoC with sandboxed sub-networks for CPU/GPU architectures," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1145–1158, 2017.

[17] N. Mansoor, P. J. S. Iruthayaraj, and A. Ganguly, "Design methodology for a robust and energy-efficient millimeter-wave wireless Network-on-Chip," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 1, pp. 33–45, 2015.

[18] X. Yu, J. Baylon, P. Wettin, D. Heo, P. Pratim Pande, and S. Mirabbasi, "Architecture and design of multi-channel millimeter-wave wireless network-on-chip," *IEEE Design & Test*, vol. 31, no. 6, 2014.

[19] S. Abadal, A. Cabellos-Aparicio, E. Alarcón, and J. Torrellas, "WiSync: An architecture for fast synchronization through on-chip wireless communication," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 3–17, 2016.

[20] W. Choi, K. Duraisamy, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 672–686, 2018.

[21] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas, "Replica: A wireless manycore for communication-intensive and approximate data," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 849–863, 2019.

[22] D. Sánchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 1, p. Article 4, 2010.

[23] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-NOC: A heterogeneous Network-on-Chip architecture for scalability and service guarantees," in *Proceedings of the International Symposium of Computer Architecture (ISCA)*, pp. 401–412, 2011.

[24] H. M. Cheema and A. Shamim, "The last barrier: On-chip antennas," *IEEE Microwave Magazine*, vol. 14, no. 1, pp. 79–91, 2013.

[25] Q. J. Gu, "THz interconnect: The last centimeter communication," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 206–215, 2015.

[26] J. Wu, A. Kodi, S. Kaya, A. Louri, and H. Xin, "Monopoles loaded with 3-D printed dielectrics for future wireless intra-chip communications," *IEEE Transactions on Antennas and Propagation*, vol. 65, no. 12, pp. 6838–6846, 2017.

[27] V. Pano, I. Tekin, I. Yilmaz, Y. Liu, K. R. Dandekar, and B. Taskin, "TSV antennas for multi-band wireless communication," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 1, pp. 100–113, 2020.

[28] H. K. Mondal, R. C. Cataldo, C. A. M. Marcon, K. Martin, S. Deb, and J.-P. Diguet, "Broadcast- and power-aware wireless NoC for barrier synchronization in parallel computing," in *Proceedings of the 31st IEEE International System-on-Chip Conference (SOCC)*, 2018.

[29] S. Abadal, C. Han, and J. M. Jornet, "Wave propagation and channel modeling in chip-scale wireless communications: A survey from millimeter-wave to terahertz and optics," *IEEE Access*, vol. 8, pp. 278–293, 2019.

[30] Y. Chen and C. Han, "Channel modeling and characterization for wireless networks-on-chip communications in the millimeter wave and terahertz bands," *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, vol. 5, no. 1, pp. 30–43, 2019.

[31] I. El Masri, T. Le Gouguec, P.-M. Martin, R. Allanic, and C. Quendo, "Electromagnetic characterization of the intra-chip propagation channel in Ka and V bands," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 9, no. 10, pp. 1931–1941, 2019.

[32] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, 1978.

[33] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: A scalable cache coherence scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[34] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[35] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," in *International Symposium on Computer Architecture*, 1989.

[36] A. Gupta and W.-D. Weber, "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Transactions on Computers*, 1992.

[37] A. Mestres, S. Abadal, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio, "A MAC protocol for reliable broadcast communications in wireless network-on-chip," in *Proceedings of the 9th International Workshop on Network on Chip Architectures*, pp. 21–26, 2016.

[38] J. Oh, M. Prvulovic, and A. Zajic, "TLSync: support for multiple fast barriers using on-chip transmission lines," in *Proceedings of the International Symposium on Computer Architecture*, pp. 105–115, 2011.

[39] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, 2011.

[40] X. Yu, H. Rashtian, and S. Mirabbasi, "An 18.7-Gb/s 60-GHz OOK demodulator in 65-nm CMOS for wireless Network-on-Chip," *IEEE Transactions on Circuits And Systems -I: Regular Papers*, vol. 62, no. 3, 2015.

[41] X. Yu, S. P. Sah, H. Rashtian, S. Mirabbasi, P. P. Pande, and D. Heo, "A 1.2-pJ/bit 16-Gb/s 60-GHz OOK transmitter in 65-nm CMOS for

wireless Network-on-Chip," *IEEE Transactions on Microwave Theory and Techniques*, vol. 62, no. 10, 2014.

[42] B. Xu, Y. Zhou, and Y. Chiu, "A 23-mW 24-GS/s 6-bit voltage-time hybrid time-interleaved ADC in 28-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, 2017.

[43] S. Saxena, G. Shu, R. K. Nandwana, M. Talegaonkar, A. Elkholy, T. Anand, W. S. Choi, and P. K. Hanumolu, "A 2.8 mW/Gb/s, 14 Gb/s serial link transceiver," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 5, 2017.

[44] F. Gutierrez, S. Agarwal, K. Parrish, and T. S. Rappaport, "On-chip integrated antenna structures in CMOS for 60 GHz WPAN systems," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 8, 2009.

[45] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[46] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," tech. rep., 2009.

[47] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT - A tool connecting emerging photonics with electronics for opto-electronic Networks-on-Chip modeling," in *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip (NOCS)*, 2012.

[48] C. W. Byeon, K. C. Eun, and C. S. Park, "A 2.65-pJ/bit 12.5-Gb/s 60-GHz OOK CMOS transmitter and receiver for proximity communications," *IEEE Transactions on Microwave Theory and Techniques*, 2020.

[49] S. Abadal, M. Iannazzo, M. Nemirovsky, A. Cabellos-Aparicio, H. Lee, and E. Alarcón, "On the area and energy scalability of wireless Network-on-Chip: A model-based benchmarked design space exploration," *IEEE/ACM Transactions on Networking*, vol. 23, no. 5, pp. 1501–1513, 2014.

[50] M. F. Chang, J. Cong, A. Kaplan, M. Naik, G. Reinman, E. Socher, and S.-W. Tam, "CMP Network-on-Chip overlaid with multi-band RF-interconnect," in *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[51] H. K. Mondal, S. Kaushik, S. H. Gade, and S. Deb, "Energy-efficient transceiver for wireless NoC," in *Proceedings of the 30th International Conference on VLSI Design and 16th International Conference on Embedded Systems (VLSID)*, 2017.

[52] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of SPLASH-2 and PARSEC," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[53] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 101–111, 2016.

[54] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 72–81, 2008.

[55] K. Duraisamy, H. Lu, P. P. Pande, and A. Kalyanaraman, "High-performance and energy-efficient Network-on-Chip architectures for graph analytics," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 26, pp. 1–26, 2016.

[56] X. Li, K. Duraisamy, J. Baylon, T. Majumder, G. Wei, P. Bogdan, D. Heo, and P. P. Pande, "A reconfigurable wireless NoC for large scale microbiome community analysis," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1653–1666, 2017.

[57] X. Li, K. Duraisamy, P. Bogdan, J. R. Doppa, and P. P. Pande, "Scalable Network-on-Chip architectures for brain–machine interface applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1895–1907, 2018.

[58] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," in *Proceedings of the 16th International Symposium on Computer Architecture (ISCA)*, pp. 2–15, 1989.

[59] M. M. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: An approach for extending SMPs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 25–36, 2000.

[60] M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 182–193, 2003.

[61] K. Strauss, X. Shen, and J. Torrellas, "Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 327–342, 2007.

[62] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects," in *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 67–78, 2009.

[63] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, "SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 25–36, 2014.

[64] W.-C. Kwon and L.-S. Peh, "A universal ordered NoC design platform for shared-memory MPSoC," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 697–704, 2015.

[65] B. K. Daya, L.-S. Peh, and A. P. Chandrakasan, "Low-power on-chip network providing guaranteed services for snoopy coherent and artificial neural network systems," in *Proceedings of the 54th Annual Design Automation Conference (DAC)*, 2017.

[66] N. Kirman, M. Kirman, R. Dokania, J. F. Martinez, A. B. Apsel, M. A. Watkins, and D. H. Albonesi, "Leveraging optical technology in future bus-based chip multiprocessors," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 492–503, 2006.

[67] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 132–143, 2011.

[68] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "ATAC: A 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 477–488, 2010.

[69] C. Batten, A. Joshi, V. Stojanovic, and K. Asanovic, "Designing chip-level nanophotonic interconnection networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 137–153, 2012.

[70] C. Thraskias, E. Lallas, N. Neumann, L. Schares, B. Offrein, R. Henker, D. Plettemeier, F. Ellinger, J. Leuthold, and I. Tomkos, "Survey of photonic and plasmonic interconnect technologies for intra-datacenter and high-performance computing communications," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 4, pp. 2758–2783, 2018.

[71] A. Carpenter, J. Hu, J. Xu, M. Huang, and H. Wu, "A case for globally shared-medium on-chip interconnect," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 271–282, 2011.

[72] A. Carpenter, J. Hu, O. Kocabas, M. Huang, and H. Wu, "Enhancing effective throughput for transmission line-based bus," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 165–176, 2012.

[73] J. Oh, A. Zajic, and M. Prvulovic, "Traffic steering between a low-latency unswitched TL ring and a high-throughput switched on-chip interconnect," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 309–318, 2013.

[74] J. W. Holloway, G. C. Dogiamis, and R. Han, "Innovations in terahertz interconnects: High-speed data transport over fully electrical terahertz waveguide links," *IEEE Microwave Magazine*, vol. 21, no. 1, pp. 35–50, 2020.

[75] B. M. Beckmann and D. A. Wood, "TLC: Transmission line caches," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pp. 43–54, IEEE, 2003.

[76] J. L. Abellán, J. Fernández, and M. E. Acacio, "Efficient hardware barrier synchronization in many-core CMPs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1453–1466, 2012.

[77] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 686–698, 2015.