WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication *

Sergi Abadal,[†‡] Albert Cabellos-Aparicio,[‡] Eduard Alarcón,[‡] and Josep Torrellas[†]

[†]Univ. of Illinois, Urbana-Champaign, IL, USA
http://iacoma.cs.uiuc.edu

[‡]Univ. Politècnica de Catalunya, Barcelona, Spain
abadal@ac.upc.edu

Abstract

In shared-memory multiprocessing, fine-grain synchronization is challenging because it requires frequent communication. As technology scaling delivers larger manycore chips, such pattern is expected to remain costly to support.

In this paper, we propose to address this challenge by using on-chip wireless communication. Each core has a transceiver and an antenna to communicate with all the other cores. This environment supports very low latency global communication. Our architecture, called *WiSync*, uses a per-core *Broadcast Memory* (BM). When a core writes to its BM, all the other 100+ BMs get updated in less than 10 processor cycles. We also use a second wireless channel with cheaper transfers to execute barriers efficiently. WiSync supports multiprogramming, virtual memory, and context switching. Our evaluation with simulations of 128-threaded kernels and 64-threaded applications shows that WiSync speeds-up synchronization substantially. Compared to using advanced conventional synchronization, WiSync attains an average speedup of nearly one order of magnitude for the kernels, and 1.12 for PARSEC and SPLASH-2.

Keywords on-chip wireless communication; synchronization

1. Introduction

In shared-memory programming, there are several common idioms that require frequent synchronization between potentially far-off processors. One is mutual exclusion to popular locations, which involves repeatedly writing and reading locks and/or other variables. Another is frequently-accessed global barriers, which involve reading and writing counts and flags. Other patterns include repeated broadcasts, reductions, and producer-consumer communications which, in addition to the data communicated, often need flags to coordinate writes and reads.

Computer architectures have traditionally struggled to support these patterns efficiently. Some machines have provided advanced hardware support, such as a barrier network in Cray T3D [12], synchronization registers in Cray T3E [38], a collectives network in Blue Gene/L [15], and fetch-and-Φ operations in the SGI Origin [25]. Also, there are many research proposals for advanced hardware support for synchronization (e.g., [5, 7, 26, 37, 40, 53]). As technology scaling delivers larger manycore chips, these patterns are expected to remain costly to support within the chip.

On-chip wireless technology provides a promising approach to address this challenge [1, 3, 13]. Suppose that each core is augmented with a transceiver (i.e., a transmitter plus a receiver) and an antenna. The transmitter modulates and radiates Radio Frequency (RF) signals through the antenna. The signals propagate throughout the chip and are received by all the receiver antennas tuned to the same frequency.

This environment provides two key supports for the idioms outlined above. First, since all antennas are listening, it naturally supports broadcast. Consequently, global communication is easily supported. Second, the latency of point-to-point communication is at least one order of magnitude lower than in current on-chip networks (i.e., around 5 processor cycles), and not dependent on the distance between source and destination.

There are existing prototypes of on-chip antennas and transceivers using conservative implementations [14, 49, 51, 52]. For example, Yu et al. [51] used standard 65 nm CMOS technology to build an antenna and transceiver that provided a bandwidth of 16 Gbit/s (enough for our purposes, as we will see) with an area of 0.23 mm^2 and a power consumption of 31.2 mW. This design is conservative — a result of the current lack of real uses for this technology beyond sensor networks. However, extrapolating this design using scaling projections [11] backed up by recent implementation trends [2, 19], it seems reasonable to use 22 nm CMOS technology to build a similar transceiver and antenna with an area of 0.1 mm^2 at 16 mW, providing the same 16 Gbit/s or perhaps higher.

In this paper, we present a novel architecture that leverages wireless communication in a large manycore to perform fast synchronization. We call it *WiSync*. The key component of WiSync is a per-core *Broadcast Memory (BM)*. Thanks to

*This work was supported in part by NSF under grants CCF-1012759, CNS-1116237, and CCF-1536795; by the Catalan Government under grant 2014SGR-1427; and by the Spanish State Ministry of Economy and Competitiveness under grant aid PCIN-2015-012.

ASPLOS '16, April 02–06, 2016, Atlanta, GA, USA.
 ISBN 978-1-4503-4091-5/16/04…$15.00.
DOI: http://dx.doi.org/10.1145/2872362.2872396.

the wireless communication, when a core writes to its BM, all the other 100+ BMs get updated in less than 10 processor cycles. WiSync uses two wireless channels: one for data transfer, and a cheaper one that transfers only tones to execute barriers very efficiently. We also present an example ISA. The resulting manycore supports multiprogramming, virtual memory, and context switching.

We evaluate WiSync with simulations of 128-threaded kernels and 64-threaded applications. The results show that WiSync speeds-up synchronization substantially. Compared to using advanced conventional synchronization, WiSync delivers an average speedup of nearly one order of magnitude for the kernels, and 1.12 for PARSEC and SPLASH-2.

2. Background

Recent advances in CMOS RF technology [18, 24, 28, 29, 36, 39] have enabled the development of antennas and transceivers that can operate in chip-scale settings. A transceiver is mainly characterized by the frequency band at which it transmits. The width of the frequency band determines the bandwidth provided by the transceiver. Specifically, using simple signal modulation, it can be shown that a given frequency band in GHz results in the same number of Gigabits per second (Gb/s) in bandwidth provided [8]. Common frequency bands for state-of-the-art on-chip wireless networks are 16-20 GHz wide, with a center at around 60 GHz. Hence, the bandwidth of the transceiver is 16-20 Gb/s.

The design of a transceiver involves tradeoffs between area, power consumption, bandwidth, and design complexity. Generally, for a given frequency point, widening the frequency band increases the area and power overheads linearly, up to a certain limit. Beyond that limit, increasing the frequency band implies using more complex and power-hungry RF components which cause a large leap in area and power. In all cases, circuit-level and system-level optimizations can help to reduce area and power without impacting upon the bandwidth.

On the other hand, operating at a higher frequency point (e.g., 90 GHz) reduces the area (since the lateral size of the passive RF components is roughly inversely proportional to the frequency), and generally makes it easier to support wider frequency bands. However, the technology at 100 GHz and higher frequencies is still exploratory.

Currently, there are implementations of on-chip antennas and transceivers working at frequency bands around 60 GHz and 90 GHz [14, 49, 51, 52]. As indicated before, the implementations are conservative because there is a lack of real uses for this technology beyond sensor networks. Yu et al. [51] used 65 nm CMOS technology to build an antenna and transceiver that provided a bandwidth of 16 Gb/s with an area of 0.23 mm^2 and a power of 31.2 mW. At this technology, a zig-zag antenna working at 60 GHz takes an area of approximately 0.02 mm^2 [14].

Using these numbers and scaling projections [11] backed up by recent implementation trends [2, 19], we envision that one could build an antenna and transceiver at 22-nm technology with an area of 0.1 mm^2 at 16 mW, providing the same 16 Gb/s. To reach these figures, we consider a sublinear area scaling, more conservative than the linear trend used in related RF interconnect works [11, 33], as well as a power reduction commensurate with the 1.67x scaling trend predicted in [11]. Alternatively, one can double the bandwidth to 32 Gb/s, with the reduced area but with little gain in power consumption over the 65 nm numbers.

Note that, in the chip, the antenna and transceiver are kept powered-on even if they are not communicating. This is done to reduce start-up/wind-down overhead. Hence, they consume about the same power while transmitting as while not transmitting.

Future Trends. Currently, both industrial and academic researchers are pushing the state-of-the-art toward miniaturization and higher frequencies [18, 19, 20, 28, 35, 48]. Abadal et al. [2] take a wide variety of transceiver designs and extrapolate area and power trends. The trends point toward increasing frequency and reducing area and power. They predict that using CMOS (beyond 22 nm), SiGe BiCMOS, or Graphene technologies at 300 GHz frequencies, one could support bandwidths of 64 Gb/s with an area in the order of 0.01 mm^2. Others [11, 24] make similar predictions, pushing the bandwidth to 100 Gb/s. Overall, we expect that, if the case for on-chip wireless communication in general-purpose computing is made, progress towards these aims will be sped-up.

Other Technologies. There are two emerging technologies that can also offer on-chip broadcast capabilities: transmission lines (TLs) [4, 5, 10, 33, 34, 41, 43] and nanophotonics [6, 21, 23, 42, 46, 47]. While these technologies are more energy-efficient and provide more bandwidth than wireless communication, they are more complicated and less scalable than wireless communication. Indeed, these technologies complicate the floorplan, as they require laying out an extra network. In terms of scalability, TLs are challenged by signal reflection, while nanophotonics by laser power requirements. We discuss them further in Section 8.

3. Overview of WiSync

3.1 Main Idea

Fine-grain synchronization has traditionally been challenging to support efficiently because it involves frequent communication between potentially far-off processors. On-chip wireless technology can address this bottleneck. If each core in a large manycore uses an antenna and a transceiver to send and receive wireless signals, we have two key properties.

First, since all antennas are listening, we naturally have broadcast (or multicast) of writes. Second, the latency of point-to-point communication is at least one order of magnitude lower than in current on-chip networks, and not dependent on the distance between source and destination. Specifically, consider broadcasting a 64-bit datum. Adding up a

special address and other bits, the message may be about 80 bits. With current technology, the wireless transfer can be done in five 1-ns cycles; with future technology, it may be doable in 2 ns. Adding to this, wireless technology does not complicate the chip layout because no additional wires are needed. Moreover, both area and power are scalable with CMOS technology.

A limitation of on-chip wireless communication is that, if we use a single data transfer channel to minimize cost, only one core can send data at a time. When the network is busy, a core has to wait; if it is free and the core attempts to send data, there may still be a collision with another core. In this case, the cycle(s) are lost and both cores have to retry. Wireless communication also introduces issues related to reliability and security — e.g., an external entity may disrupt the communication unintentionally or on purpose. In this paper, we do not consider these issues.

WiSync abstracts the wireless capability in the form of a per-core *Broadcast Memory (BM)* that is used by variables declared of type *broadcast* in the program. All BMs are connected to the wireless network, and they all contain the exact same, *replicated* variables. When a core writes to a location in its BM, all the other BMs get automatically updated. There is a total order of writes to BMs across all cores.

Figure 1 shows this idea. Variables x and y are replicated in all the BMs. Three concurrent writes by different processors can result in different interleavings, but no update is lost and all processors observe the same write interleaving. The jagged line means that the non-local BMs are physically far.

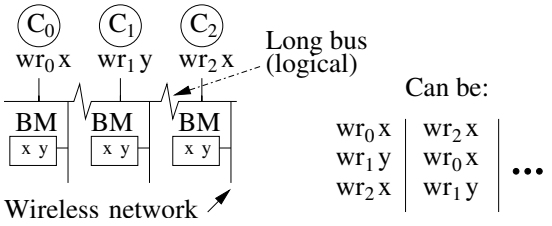


Figure 1: How wireless writes appear in WiSync.

The operating system manages the BM resource with virtual memory, while the program addresses the BM transparently. As a result, WiSync supports multiprogramming — it is likely that a large manycore will be shared by multiple applications — context switching, process migration, and memory access protection.

3.2 WiSync Architecture

Figure 2 shows the WiSync architecture. Each node in the manycore contains: a core with its L1 and L2 caches, a transceiver, two antennas, the BM, and two special bits. The second antenna is for a special, inexpensive *Tone* channel that we discuss later, and adds a very small area and power overhead. The transceiver has three modules: physical layer (PHY), Medium Access Control (MAC), and the Tone Controller. The PHY module interfaces with the antennas by serializing and modulating the data to transmit, detecting the transmission collisions, and demodulating and deserializing the received data. The MAC module determines when a write should be transmitted through the wireless network, and manages the collisions.

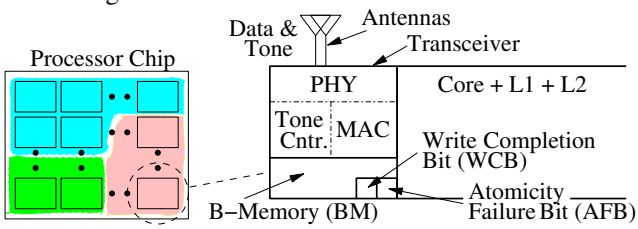


Figure 2: WiSync architecture. The different colors represent different programs running on the chip.

To understand the second antenna and the tone controller, note that, to keep the energy and area footprint of the wireless hardware low, WiSync transfers data in a single frequency band. However, we will see that it is beneficial to add an additional channel that only supports sending a tone. The tone controller manages it.

The BM stores the variables that are communicated through the wireless network. These variables are declared as *broadcast variables* in the program, and are kept replicated in all the BMs. They are uncacheable in the core's L1-L2 cache hierarchy. The core accesses its BM with plain loads and stores, using a range of virtual addresses that are translated in the TLB, but that bypass the L1 and L2 caches. The regular variables use the caches, which are kept coherent with an ordinary cache coherence protocol.

A plain load instruction can read from an address in the local BM into a register. A plain store instruction can store the value of a register into an address in the local BM and (thanks to the wireless network) in *all the BMs*. The write may collide in the wireless network, in which case no BM (including the local one) is updated. The MAC module keeps retrying the wireless transmission until it succeeds, at which point the local BM is also updated.

To reduce overhead, WiSync has *Bulk* load and store instructions. They take a register R and a BM address *BM_addr*. A Bulk load loads from four consecutive locations in the local BM starting at *BM_addr* into four registers starting at R. A Bulk store stores the four registers to four consecutive locations starting at *BM_addr* in the local and the remote BMs.

WiSync supports atomic read-modify-write (RMW) instructions to the BM, such as Test&Set, Fetch&Inc, Fetch& Add, and Compare-and-Swap (CAS). A RMW instruction involves reading a local BM location, bringing the data to the pipeline, updating the data in the pipeline, and then writing the result to the local BM and (using the wireless network) to all the BMs. To succeed, the instruction must guarantee atomicity from the time it reads from the BM until all the BMs are updated, possibly after several collisions and

retries. To provide correct operation, the BM controller has two bits called Write Completion Bit (WCB) and Atomicity Failure Bit (AFB) (Figure 2).

Loads, stores, and RMW instructions can operate on different-sized data, although we will focus on 64-bit operations here.

As shown in Figure 2, multiple programs can be running concurrently and trying to use the BM and wireless network at the same time. Conventional access-bit permissions in the TLB ensure that programs access only their own data.

4. WiSync Architecture

4.1 Organization of the Wireless Transfer

A wireless system can be set up to use a single channel or multiple channels at different frequency ranges. Supporting multiple channels enables parallel wireless communication that can be exploited both within a program and across programs. However, it adds complication. Since we want to keep our system simple and the transceiver small, we choose to use a single channel.

We time-slot the channel in 1-ns slots — equal to the clock period of our 1GHz cores. Typically, we want to use the wireless network for single writes, such as a write to a reduction variable or to a lock. Hence, a transfer involves: a 64-bit datum, its address (11 bits, as we will see), a *Bulk* bit (to denote whether this is the first datum of a Bulk transfer), and a *Tone* bit (to denote whether this is for the tone channel, as we will see). The total size of a transfer is 77 bits.

As discussed in Section 2, current transceivers can transmit at 16-32 Gb/s [51], although in the next few years, this number may go up [24]. Hence, we use 4 cycles to transfer the 77 bits, which gives a required wireless bandwidth of 77b/4ns≈19Gb/s. This is a conservative value [51].

To keep the transceiver simple, WiSync takes 5 cycles for the actual transmission. The second cycle is used for the transceiver to listen if there was a collision in the first cycle. If there was a collision, the transfer is aborted, and the channel is free in the third cycle. Otherwise, the next three cycles send the rest of the message with guaranteed no collision. In this way, the penalty of a collision is 2 cycles instead of 5 cycles. A transmitter-receiver switch capable of working at these speeds has been demonstrated [45].

As per Section 2, a 19 Gb/s bandwidth requires a transmission frequency band of 19 GHz. We choose the central frequency to be 60 GHz, which can be supported with a small transceiver [2]. As shown in Figure 3, we call the resulting channel the Data channel. A transceiver and antenna for a similar frequency range have been built, and shown to consume 31.2 mW [51]. It is widely expected that technology advances will reduce this power.

To help message processing, every wireless message carries the Bulk bit in its first cycle. In ordinary accesses, the bit is zero; in Bulk accesses, it is set. When a transceiver observes the first cycle of a message, it knows the length of the message: if the Bulk bit is zero, the message will take 5 cycles; otherwise it will take 15 cycles.

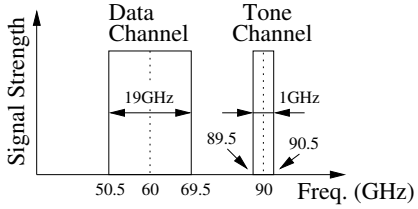


Figure 3: Transmission channels in WiSync.

The reason why a Bulk message only takes 15 cycles rather than 4×5=20 cycles is because we optimize it. Specifically, the second, third, and fourth word of a Bulk message do not need the collision check that occurs in the second cycle of an ordinary message. In addition, they do not need to carry an address, Bulk bit or Tone bit. Hence, given the network bandwidth that we use, these three words transfer their 3×64=192 bits in 10 cycles. The complete Bulk message then takes 5+10=15 cycles to transfer.

When the wireless network is busy, a transceiver ready to send a message waits to send it until the cycle when the network is next expected to be free. This is 15 or 5 cycles after the last message started (depending on whether or not that message was a Bulk one), or after the second cycle if a collision is detected.

In a collision, the MAC module in each of the colliding nodes stops the transmission, discards the messed-up information, and prevents the store from updating the local BM. It then retries the transmission. To maximize the use of the network, the MAC module uses a retry algorithm that temporarily reduces its assertiveness when there is contention. WiSync uses the well-known exponential backoff algorithm [27, 31, 32], which progressively increases the waiting period on a collision.

To reduce contention for the Data channel, WiSync provides an additional channel called the *Tone channel* that offloads the common barrier synchronization pattern. We apply the proposal of Oh et al. [33] for transmission lines, to wireless networks. In the Tone channel, nodes send a tone rather than data.

Specifically, the first core that reaches the barrier sends a message through the Data channel with the Tone bit set. On receiving the message, all the other nodes respond with a continuous tone in the Tone channel, creating repeated collisions. Then, as soon as a core reaches the barrier, its transceiver stops sending the tone. Hence, when the tone finally disappears, all cores have arrived at the barrier.

The Tone channel is also slotted in 1 ns slots. Each message is 1 bit, for a resulting transmission rate of 1 Gb/s. This means that the frequency band of the Tone channel is only 1 GHz. With such a short band, the power needed to support it (i.e., the transceiver extensions and a second antenna) is very small. With the same conservative assumptions we have been using, we estimate it adds 2 mW.

We place this channel at 90 GHz (Figure 3), far from the Data channel to avoid interference. At these high frequencies, signal attenuation is higher, but this is no problem because the Tone channel only transfers a tone rather than data.

4.2 B-Memory Operation and Interface

The BM in a node contains space for all the allocated variables that are declared as *broadcast* type in the programs currently running on the chip. Such variables are replicated in all the BMs and their values are kept consistent at all times. To use the BM efficiently, each BM entry is 64 bits rather than a line. Moreover, for protection, each entry is tagged with the PID of the program or process that allocated it.

When a program allocates a broadcast variable, the variable is allocated in all the BMs, and is tagged with the correct PID. When a core reads or writes a BM entry, the PID is checked in hardware to ensure the action is legal (Section 4.4 describes the address translation). When a variable is deallocated, it is removed from all the BMs.

The optimal size of the BM in each node is likely to be small, e.g., four 4-KB pages. The reason is two-fold. First, large memories require many address bits to be included in each wireless message. With 16KB, we already need 11 bits. Secondly, programs would not usually declare many broadcast variables. If the BM runs out of space for a variable, we envision transparently allocating the variable in a page of regular memory, and access it through the wired network.

4.2.1 Interface to Basic BM Instructions

Loads access the local BM and always succeed. For stores and RMW instructions, WiSync has the *Write Completion Bit (WCB)* and the *Atomicity Failure Bit (AFB)* (Figure 2). These bits are set/reset in hardware and accessible to the software through a register.

The WCB is associated with a store or RMW instruction. It is set when the update operation completes — both the global broadcast and the local BM update. In certain situations, the OS or the user code may benefit from this information. The AFB is associated with a RMW instruction. It is set if its atomicity fails — in which case the instruction completes without the BM write.

When a core executes a store instruction, the transceiver first attempts to broadcast the update to all the remote BMs. If there is a network collision, the transceiver keeps retrying until it succeeds. After it succeeds, the local BM is updated, the WCB gets set, and the pipeline receives the acknowledgment that the store is performed. No subsequent store from the local core can proceed to the global broadcast until the current one has performed all the steps above. Subsequent loads from the local core may or may not be allowed to read any BM address while the current write is in progress, depending on the memory consistency model desired. If loads are allowed to, we have a TSO memory model; if they are not allowed to read any BM address beyond the one being written, we have a sequential consistency model.

When a core executes a RMW instruction, the hardware reads the datum from the local BM into the pipeline, updates the datum in the pipeline, and then tries to write the datum to the BM. As usual, the write involves performing the global broadcast first and, when it succeeds, updating the local BM. It is possible that, after the read from the local BM and before the write manages to get to the wireless network, a remote node updates the variable in the local BM. This is detected in hardware by comparing the addresses of incoming stores to those of pending RMW accesses. In this case, the atomicity of the instruction has failed. As soon as this occurs, the AFB bit gets set. Later, when the write is attempted, it fails — i.e., the RMW instruction neither broadcasts its value nor it updates the local BM. At that time, the WCB gets set because the RMW operation has terminated.

Consequently, a RMW instruction needs to be followed by a software check of the AFB bit. The instruction has executed atomically and, therefore, performed the write, only if AFB=0. If, instead, AFB=1, the write never occurred, and the RMW instruction has to be re-executed. Note that this discussion assumes that the hardware does not allow a load following the RMW instruction to read the AFB register until after the RMW instruction has finished. An exception between the RMW instruction and the AFB check before the wireless transfer has succeeded automatically sets AFB and aborts the wireless transfer. AFB is saved and restored on context switch.

4.2.2 Interface to Tone Channel Instructions

A core has special BM load and store instructions that enable the use of the Tone channel for a particular BM address. They are *tone_ld R, BM_addr* and *tone_st R, BM_addr*.

Recall that we use the Tone channel to implement a barrier: each core needs to tell all the others when it has arrived at the barrier. *Tone_ld* and *tone_st* provide a very efficient implementation. Specifically, when a core reaches the barrier, it performs a *tone_st* operation on the BM location. Note that this is not an ordinary update of the location. Then, the core keeps reading the BM location using *tone_ld*. The load will return a special code when all the participating cores have performed the *tone_st*. In the meantime, the core may choose to do other work, while periodically polling with *tone_ld*.

The implementation of these instructions relies on the tone controller as follows. On a *tone_st*, the hardware does not update the BM location. Instead, the tone controller checks whether it is currently issuing a tone for this address in the Tone channel. If so, the local core is not the first core to arrive at the barrier, and the controller stops issuing the tone; otherwise, the local core is the first one to arrive, and the controller sends a message in the Data channel with the address of the BM location and the Tone bit *set*. The content of the 64-bit data field is immaterial.

The tone controller in any node also performs two actions based on incoming events. First, when it receives a message in the Data channel with the Tone bit set, if the local core

```
retry: fetch&inc R, BM_addr
       if (AFB) {
         jmp retry
       }
       else {
         /* success */
       }
```
(a)

```
retry: ld R_old, BM_addr
       . . . .
       if (!CAS(BM_addr, R_old, R_new)) {
         jmp retry  /* comparison already failed */
       }
       else {
         if (AFB) {
           jmp retry   /* atomicity failure */
         }
         else {
           /* success */
         }
       }
```
(b)

```
for ( ) {
  /* work */
  barrier: local sense = !local_sense
           tone_st (addr)
           spin until (tone_ld(addr)==local_sense)
}
```
(c)

Broadcast Writer

```
for ( ) {
  local_sense = !local_sense
  write data
  count = N
  release = local_sense
  spin until (count==0)
}
```

Broadcast Readers

```
for ( ) {
  local_sense = !local_sense
  spin until (release==local_sense)
  read data
  fetch&add (count,−1)
}
```
(d)

Figure 4: Examples of code used for synchronization.

participates in the barrier, it starts issuing a continuous tone in the Tone channel. This will cause repeated collisions with other nodes, which is precisely what we want. We will see in Section 4.4 how barrier participation is detected.

The second action of the tone controller occurs as soon as it detects that the Tone channel has fallen silent. At this point, all the cores have reached the barrier. Then, the controller toggles the value of the local BM location. Such location can only take the values zero or non-zero.

This is a sense-reversing barrier [16]. When a core spinning with *tone_ld* on that address observes that the value has changed, it knows that the barrier has ended. Later, we show how to support multiple concurrent tone barriers.

4.3 Supporting Synchronization Operations

We now outline how WiSync supports some of the popular synchronization operations.

4.3.1 Basic Read-Modify-Write Primitives

Figure 4(a) shows the pseudo-code of the algorithm used to execute a basic RMW operation. The example uses fetch&increment. After the core executes the RMW instruction, it checks the AFB register bit. If it is set, the instruction has been aborted and it has to be retried. Otherwise, the operation performed successfully. Similar pseudo-code is used for test&set and fetch&add.

Figure 4(b) shows pseudo-code for CAS. The code assumes that a CAS returns zero if the content of the BM location is different from the CAS' second argument; otherwise, it returns non-zero. Even if it returns non-zero, however, the CAS may not have executed atomically (and failed to perform the wireless broadcast). Hence, the code checks the AFB register. If it is set, the CAS needs to be retried.

4.3.2 Barriers Using the Data Channel

There are two types of barriers: AND-barriers and OR-barriers. The former are the conventional ones, where a processor arriving at a barrier waits until all other processors have also arrived. For them, we use the popular sense-reversing barrier algorithm [16] with fetch&increment. To save BM space, a single 64-bit BM entry could contain the *Count* variable in one 32-bit word and the *Release* flag in the other word.

OR-barriers (also called Eurekas) are triggered as soon as one of the participating processors detects a certain condition, e.g., overflow of a variable, the solution of a parallel search, or an exception. We implement them as a boolean variable in a BM location. All processors periodically read it; when one detects the condition, it changes the variable. We use a sense-reversing implementation to allow barrier reuse.

4.3.3 Tone Barriers

AND-barriers are supported more efficiently with the Tone channel. As per Section 4.2.2, each core executes the pseudo-code algorithm of Figure 4(c). It implements a sense-reversing barrier with minimal communication. On arrival, the first core sends a message through the Data channel, while later cores simply remove their tone from the Tone channel. Then, all cores spin in their local BM location. When the Tone channel falls silent, the controller toggles the location in all the BMs, which releases all cores. This is a scalable implementation.

4.3.4 Producer-Consumer Operation

To support the single producer, single consumer pattern, we use a BM address for the data and a BM address for a flag. The producer writes the data to the data address, sets the flag, and then spins on the flag until it is cleared. The consumer spins on the flag until it is set, reads the data, and then clears the flag. The process repeats. Often, producer and consumer will use bulk stores and bulk loads, respectively. These instructions trigger reads and uninterruptible writes of multiple BM addresses at a time.

4.3.5 Reduction and Multicast/Broadcast

The BM supports reductions very efficiently. For instance, when all the cores need to add to a single variable, they can use *fetch&add R, BM_addr*. To support different types of reductions, one can include other *fetch&*Φ instructions and,

for scientific computations, floating-point versions of them. Very tight reduction loops are supported efficiently.

Multicast/broadcast is the single producer, multiple consumers pattern, and is also supported very efficiently. The producer writes to a BM address and all consumers read from it. To provide ordering between the writer and the readers, we can use an extension of the full-empty flag discussed above. Specifically, we use two additional BM addresses (or store two variables in different halves of the same BM address). One variable is a count, and the other a toggling flag, effectively implementing a sense-reversing barrier.

The pseudo-code is Figure 4(d). The producer writes the data, sets the count to *N* (the number of readers), toggles the flag, and then spins on the count until it is zero. Each reader spins on the flag until it toggles, reads the data, and then decrements the flag with *fetch&add*. The process repeats.

4.4 BM Entry Allocation and Protection

To access the BM, WiSync uses TLB-based address translation, so that programs do not have to manage memory and can benefit from access protection. However, traditional page-level assignment is suboptimal, as the BM is small and cannot afford the memory wastage due to page-level fragmentation when multiple programs use the BM. Indeed, if each program allocated a single broadcast variable but was assigned a full page, the BM would soon run out of space.

To address this problem, WiSync uses page-level TLB translation but lets different programs use different chunks of the same physical BM page — tagging each chunk with the PID of the program that owns it. Different programs have virtual pages mapped to the same physical BM page, but each program only uses its own, non-overlapping chunks of the physical page.

The smaller the chunk size is, the better the page can be utilized, and there is less fragmentation. However, there is more bookkeeping and tag overhead. In this paper, we do not examine these tradeoffs. Rather, we reckon that using BM effectively is important and, hence, use 64-bit chunks.

When a core allocates an entry in the local and remote BMs, the entry is tagged in hardware with the program's PID. When a core accesses the BM, the address is first translated. Then, at the target BM location, the program's PID is compared to the PID tag (Figure 5). A mismatch is a protection violation.

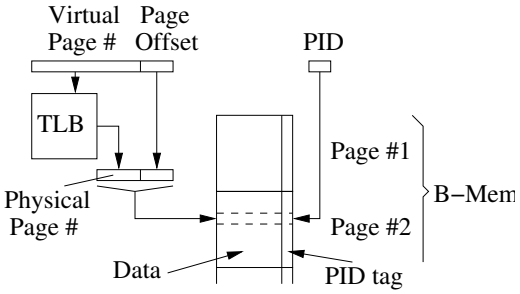


Figure 5: Broadcast memory address translation.

To allocate an entry in the BM, the core uses a special allocation instruction. The instruction broadcasts a message in the wireless network that contains the address (11 bits), PID (e.g., 8 bits), and a few miscellaneous bits. On reception, each node in the chip allocates an entry at the same address in its local BM and tags it with the PID. When this operation succeeds, the local BM allocates and tags an entry at the same address. While it appears inefficient that a variable takes space in all the BMs (even in the BMs of cores that do not run the relevant program), we do it for simplicity.

The allocation of a variable that uses the Tone Channel proceeds similarly. However, in addition, the OS in the receiving nodes records whether or not this variable is locally *armed*. Armed means that there is (or will be) a thread running on the local core that will participate in the tone barrier protocol for the variable. Later, when the node receives a message initiating a tone barrier for a variable, if the variable is locally armed, the tone controller will participate by starting a tone.

This discussion illustrates a restriction in the usage of tone barriers. Unlike in conventional barriers, when a tone barrier is allocated, the runtime needs to know which cores will participate. This is because most of the tone barrier operation is in hardware. If core participation cannot be determined until later on in the program, a tone barrier cannot be used; instead, a Data channel barrier should be used.

5. Implementation Issues

5.1 Sharing the Tone Channel

At any given time, there may be multiple tone barriers (each corresponding to a different BM address) that want to use the Tone channel. WiSync responds by time-multiplexing the Tone channel. Specifically, the Tone channel is slotted with 1-ns slots, and the slots are assigned round-robin to the currently-active tone barriers. A tone barrier is *Active* from the time the first core arrives until the last participating one arrives. Figure 6(a) shows how the slots in the Tone channel are assigned when there are 1, 2, or 3 active tone barriers.

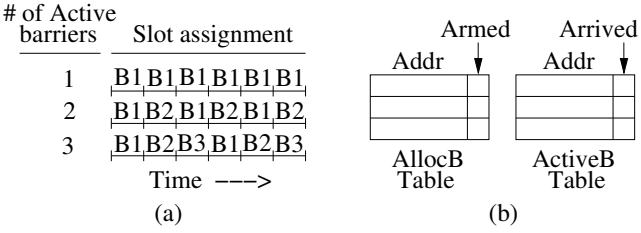


Figure 6: Sharing the Tone channel among several barriers.

To share the Tone channel, the tone controller in each node keeps two tables (Figure 6(b)): one with the *allocated* tone barriers (*AllocB*) and one with the *active* tone barriers (*ActiveB*). These tables contain the same addresses and in the same order in all the nodes of the chip. Such chip-wide consistency is required for correct assignment of slots to

barriers. It is easy to support thanks to the wireless broadcast capabilities.

In AllocB, each entry has the BM address of a tone barrier variable plus the local *Armed* bit. When a tone barrier variable is allocated by a program (Section 4.4), an entry is created in the AllocB of all the nodes in the chip. It is also at this point that the OS in each node sets the *Armed* bit to either 1 or 0, as discussed in Section 4.4. Finally, when the tone barrier variable is deallocated (possibly at program termination), its entry is removed from the AllocB of all nodes. At that time, all the entries lower in the table are shifted up.

In ActiveB, each entry has the BM address of the tone barrier variable plus an *Arrived* bit, which indicates whether the local core has arrived at the barrier. The order of the entries in this table determines the round-robin assignment of Tone channel slots to the different active tone barriers.

For a given tone barrier *B*, when the first core arrives at the barrier, the entry for *B* in AllocB is copied to the lowest position in ActiveB. This is done in all the cores. Specifically, if the first-arriving core is the local one, the usual message is sent through the Data channel, the copy is made, and then the *Arrived* bit in the copied entry is set. Instead, if the first-arriving core was a remote one, when the message that it placed in the Data channel is received, the copy is made. Then, if the local *Armed* bit was clear, the *Arrived* bit in the copied entry is set, and no tone is issued in the Tone channel. Effectively, this node refuses to participate in the barrier. On the other hand, if the local *Armed* bit was set, the *Arrived* bit in the copied entry is left clear, and the hardware starts issuing the tone in the Tone channel — but only during the assigned slots. Later, when the local processor arrives, the *Arrived* bit gets set and the local tone terminates.

At any slot assigned to this active barrier, the local tone controller uses the following algorithm. If the *Arrived* bit is zero, issue the tone. Otherwise, listen for the tone; if there is no tone, all cores have arrived. Hence, remove *B*'s entry from ActiveB and shift all the lower entries up.

AllocB and ActiveB are identical across all nodes except for the *Armed* and *Arrived* bits. As a result, all cores know the assignment of slots to individual barriers at all times. Note that the copying of an entry from AllocB to ActiveB starts as soon as the first cycle of the message transfer in the Data channel. The latency of the copy is hidden by the subsequent cycles of the message transmission.

We size AllocB and ActiveB equally, and return an error if an allocation overflows AllocB. In a multiprogramming environment, the OS has to prevent a process from starving out all other processes by using up all the AllocB entries; this can be done by limiting the per-process use of AllocB.

5.2 Context Switching and Thread Migration

The Data channel in WiSync is designed to operate correctly under context switching, thread migration, and multiple threads sharing a core. Consider first a running thread that gets preempted. Even while the thread is preempted, updates from other cores to broadcast variables will reach the local BM and update it. When the thread is rescheduled again, it will see the correct BM state.

A thread can also migrate to another core and resume execution there seamlessly. This is because the state of the BMs is identical in all the nodes. The non-BM state is not relevant to our discussion, as we assume that the cache coherence keeps it coherent. Finally, multiple threads can share a core, and update the same or different BM variables.

The situation is different for programs that use the Tone channel. The reason is that this channel is hardware-managed. In this case, threads can still be preempted, but cannot migrate or share the core with another thread that also uses the same Tone barrier. Indeed, when a tone barrier is allocated, the OS arms or disarms the local AllocB entry, depending on whether a local thread will participate. Migrating a thread would require somehow migrating this state, which is costly. Also, two threads on the same core trying to use the same tone barrier would result in incorrect operation.

5.3 Adaptively Dealing with Contention

Our collision resolution algorithm uses exponential backoff. On a collision, the transmitters back-off for a random number of cycles, whose range increases exponentially with the number of retries. In our design, the range is between 0 and 2^i-1 cycles, where i is a number incremented at every collision and decremented at every successful transmission.

It is possible to use more advanced policies with adaptivity based on the observed contention — a concept analogous to Reactive Synchronization [27]. Adaptive algorithms are not generally used in conventional non-broadcast wireless networks because it is hard to get consensus on the decisions. However, in our scenario, they would be easy to support because all nodes have all the information at all times. In our work, we have not explored such techniques.

6. Evaluation Environment

We use cycle-level execution-driven simulations to model a manycore with 16–256 cores, either with or without WiSync support. The default core count is 64. We use the Multi2sim simulator [44]. Table 1 shows the general parameters of the architecture and, in its lower part, those related to WiSync.

We compare four manycore configurations, as shown in Table 2. *Baseline* is a plain manycore with no wireless hardware. For synchronization, it uses CAS and a sense-reversing centralized barrier. *Baseline+* enhances *Baseline* hardware with: (1) virtual tree-based broadcast in the on-chip network with flit replication at the router crossbars [22], (2) MCS locks [31], and (3) Tournament barriers [31]. *WiSyncNoT* is *WiSync* without the Tone barrier support.

We run two sets of synchronization-intensive kernels and a set of applications (Table 3). The first set of kernels con-

General Parameters	
Architecture	22nm manycore with 16–256 cores (default: 64)
Core	Out of order, 2-issue wide, 1GHz, x86 ISA
ROB; ld/st queue	64 entries; 20 entries
L1 cache	Private 32KB WB, 2-way, 2-cycle RT, 64B lines
L2 cache	Shared with per-core 512KB WB banks
L2 bank	8-way, 6-cycle RT (local), 64B lines
Cache coherence	MOESI directory based
On-chip network	2D-mesh, 4 cycles/hop, 128-bit links
Off-chip memory	Connected to 4 mem controllers, 110-cycle RT
WiSync Parameters	
Per-core BM	16KB, 2-cycle RT, 64-bit wide entry
Tone channel	1Gb/s; 1-cycle transfer latency
Data Tran. channel	19Gb/s; 5-cyc transfer lat; collision detec. cyc 2
Collision handling	Exponential backoff
Transceiv+2Anten	Area: 0.12mm^2; power: 18mW

Table 1: Architecture modeled. RT means round trip.

Config.	BM?	Broadcast HW	Locks	Barriers
Baseline	No	No	CAS	Centralized
Baseline+	No	Virtual Tree	MCS	Tournament
WiSyncNoT	Yes	Wireless	Wireless	Wireless (Data only)
WiSync	Yes	Wireless	Wireless	Wireless (Data+Tone)

Table 2: Architecture configurations compared.

sists of four loops that execute barriers. They include a tight loop that we wrote (*TightLoop*) and Livermore loops 2, 3, and 6 [30]. In TightLoop, each thread adds-up the contents of a 50-element array into a local variable and then synchronizes in a barrier. The process repeats in a loop. For the Livermore loops, we focus on only these three because, as Sampson et al. [37] argue, they are the representative ones with regard to synchronization. We parallelize them and align their data as these authors do.

Barrier Kernels	TightLoop, Livermore loops 2, 3, 6
CAS Kernels	FIFO, LIFO, ADD
Application Suites	SPLASH-2, PARSEC

Table 3: Kernels and applications executed.

The second set of kernels consists of three kernels that execute CAS operations on lock-free data structures. In the ADD kernel, there is a shared queue to which, through a CAS, threads attempt to insert nodes taken from their private memory pools. A given number of instructions are executed between successive node insertions. In the FIFO and LIFO kernels, threads both enqueue and dequeue nodes from the shared queue or stack, respectively. Similarly, a given number of instructions are executed between accesses.

The applications are the entire SPLASH-2 [50] and PARSEC [9] suites. We use the standard input set sizes in SPLASH-2 and *simsmall* in PARSEC. *Dedup* and *fluidanimate* declare arrays of locks larger than the 16KB BM used here. In these cases, we allocate the first 16KB in the BM and the rest in plain memory. Finally, we modified the OpenMP libraries in order to evaluate *freqmine*.

7. Evaluation

7.1 Area and Power Comparison

Table 4 compares the estimated area and power consumption of the transceiver plus the two antennas to two popular 22 nm cores, namely the high performance Xeon Haswell and the energy-efficient Atom Silvermont. The TDP (Thermal Design Power) of an 18-core Haswell chip at 2.1 GHz is 135 W [17]. Correcting for frequency, we roughly estimate a per-core 5 W TDP. The TDP of an 8-core Silvermont chip (Avaton) at 1.7 GHz is 12 W [17]. At 1 GHz, we estimate a per-core 1 W TDP. Area numbers are from the literature.

To estimate the area and power of the transceiver plus the two antennas, we use the 22 nm 60 GHz figures outlined in Section 2. Specifically, a transceiver and one antenna consume 0.1 mm^2 and 16 mW. We then augment the transceiver with circuitry to support the tone channel, and add a second antenna at 90 GHz for the tone channel. Scaling from the 65 nm figures in [14, 49] to 22 nm, we estimate that this additional capability costs 0.04 mm^2 and 2 mW. Hence, the total cost of the transceiver plus the two antennas is 0.14 mm^2 and 18 mW (Table 4).

	Xeon Haswell Core	Atom Silvermont Core	T+2A	(T+2A) /Xeon (%)	(T+2A) /Atom (%)
Area	21.1mm^2	2.5mm^2	0.14mm^2	0.7	5.6
TDP	≈5W	≈1W	18mW	0.4	1.8

Table 4: Comparing two popular cores to the transceiver plus two antennas (T+2A), all at 22 nm.

Overall, we see that the transceiver and two antennas have a small footprint. They account for 0.7% of the area and 0.4% of the power of a Haswell core, and 5.6% of the area and 1.8% of the power of an Atom core.

7.2 Barrier Synchronization Evaluation

We first evaluate the performance of the different barrier configurations using TightLoop, which represents a very demanding environment. Figure 7 shows the number of cycles that each iteration of the loop takes on the different architecture configurations as we change the number of cores from 16 to 256. Note that the Y-axis in this plot and most of the others is *logarithmic*.

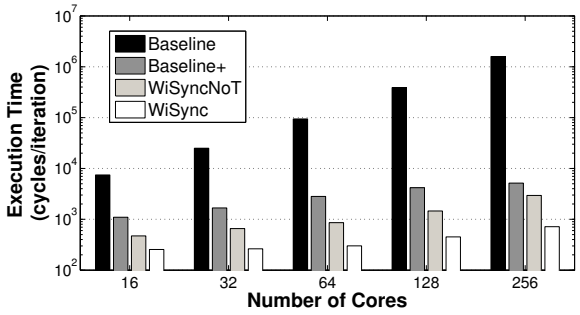
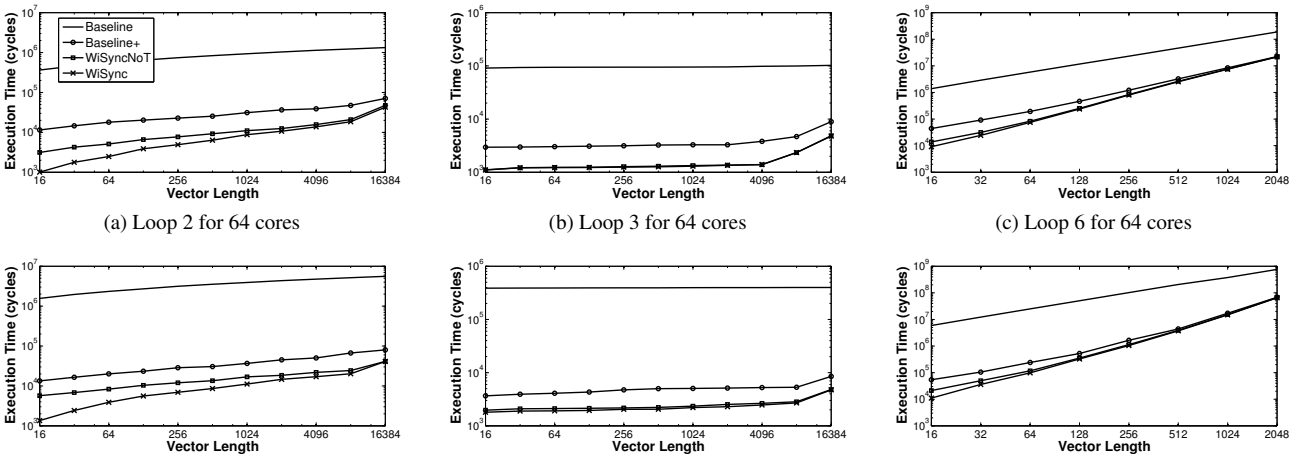


Figure 7: Execution time of TightLoop on different architecture configurations. Note that the Y-axis is *logarithmic*.



(a) Loop 2 for 64 cores (b) Loop 3 for 64 cores (c) Loop 6 for 64 cores

(d) Loop 2 for 128 cores (e) Loop 3 for 128 cores (f) Loop 6 for 128 cores

Figure 8: Execution time of Livermore loops on different architecture configurations for several vector sizes and core counts.

The figure shows a large difference between the configurations, especially for high core counts. As we increase the core count, the execution time of WiSync remains low, thanks to the Tone channel. WiSync's execution time is about one order of magnitude lower than Baseline+ (which uses a Tournament barrier). It is two to three orders of magnitude lower than Baseline.

WiSyncNoT takes 2–6x longer than WiSync to execute because of collisions in the Data channel. Hence, the Tone channel is useful for barrier-intensive codes. Still, WiSyncNoT's execution time is 2x–4x lower than Baseline+.

Figure 8 shows the execution time of Livermore loops 2, 3, and 6 on the different architecture configurations as we change the vector length. The top and bottom plots correspond to 64-core and 128-core executions, respectively.

Starting with the three upper plots, we see that WiSyncNoT and WiSync are several times faster than Baseline+, and two orders of magnitude faster than Baseline. The gains are highest with small vector lengths, where the overhead of the barrier relative to the computation is most significant. As the vector lengths increase, the computation time becomes higher, and the barrier time (including any collisions) becomes relatively less important. As a result, Baseline+ tends to get closer to WiSyncNoT and WiSync. This is especially the case for Loop 6, which has a large loop body.

We see that WiSync is significantly faster than WiSyncNoT in Loop 2 for modest problem sizes. The reason is similar to that in TightLoop: a burst of arrivals causes collisions in the Data channel in WiSyncNoT which, due to the modest duration of the loop, have a significant impact on latency. WiSync eliminates the collisions by using the Tone channel.

If we examine the lower plots, we see a wider gap between Baseline and the rest of configurations for 128 cores. The difference between WiSync, WiSyncNoT, and Baseline+ follows similar trends as for 64 cores.

Overall, we conclude that, while WiSync's Data channel does well across the board, the Tone channel is even better for some workloads.

7.3 CAS Synchronization Evaluation

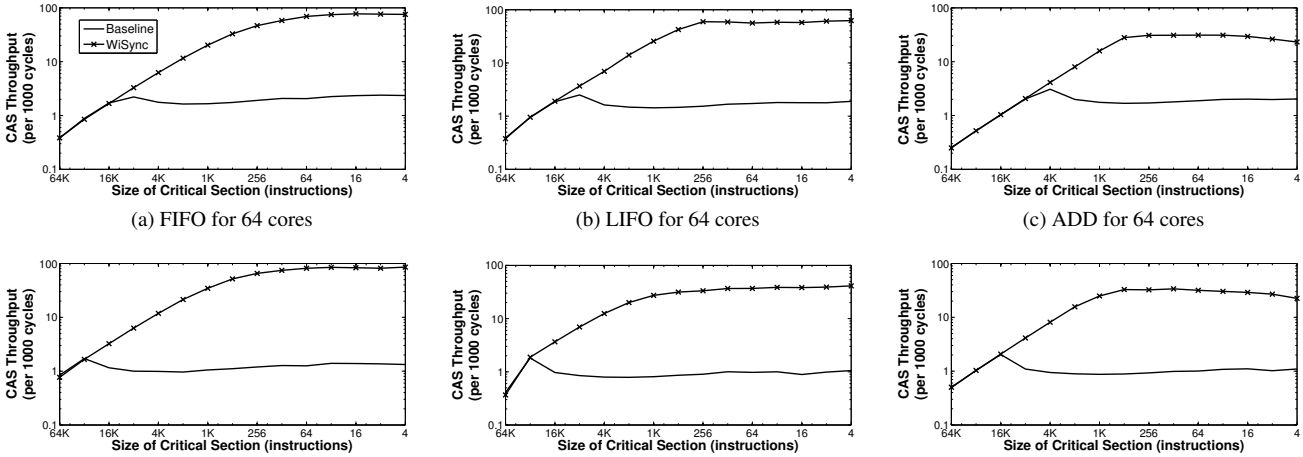
We now consider kernels with CAS operations and compare executions using the wireless network and using conventional systems. Since these kernels involve lock-free structures and do not use barriers, the results are independent of the lock and barrier implementation. Consequently, we simply compare WiSync (where CASes use the BM) to Baseline (where CASes use the cache hierarchy).

Figure 9 shows the throughput of the FIFO, LIFO, and ADD CAS kernels on the two architecture configurations as we change the number of instructions that a given processor executes between consecutive CASes. This is shown in the X-axes as critical section size, and small numbers are to the right. The plots show the number of successful CASes per 1,000 cycles. The top and bottom plots correspond to 64-core and 128-core executions, respectively.

The figures show that WiSync is able to attain a much higher CAS throughput than a conventional architecture. For 64 cores, there is little or no difference between the architectures when the number of instructions between CASes is 8-16K or larger. However, the difference increases as the critical section becomes smaller and contention rises. By the time we have about 2K instructions, WiSync delivers a CAS throughput that is about one order of magnitude higher than Baseline. With 128 cores, at about 4K instructions in the critical section, WiSync delivers a one order of magnitude higher CAS throughput than Baseline.

7.4 Full Application Evaluation

We now consider the entire PARSEC and SPLASH-2 application suites running with 64 cores. Figure 10 shows the speedup of Baseline+, WiSyncNoT, and WiSync over the Baseline architecture for all the applications. The two right-



(a) FIFO for 64 cores (b) LIFO for 64 cores (c) ADD for 64 cores

(d) FIFO for 128 cores (e) LIFO for 128 cores (f) ADD for 128 cores

Figure 9: CAS throughput of three kernels on different architecture configurations for several critical section sizes and core counts. In the charts, higher is better.

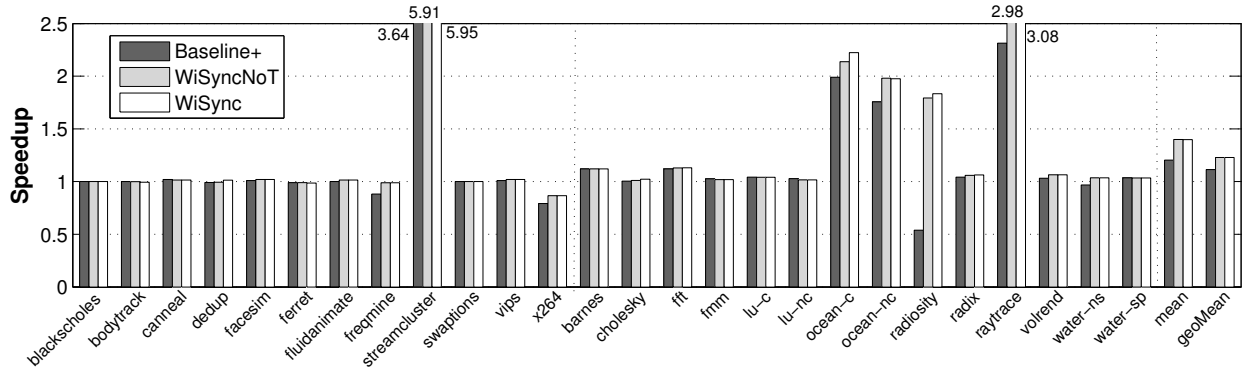


Figure 10: Speedup of different architecture configurations over Baseline running PARSEC and SPLASH-2 for 64 cores.

most set of bars show the arithmetic and the geometric mean, respectively.

Based on the geometric mean, WiSync delivers an average speedup of 1.23 over Baseline. Moreover, compared to the more advanced Baseline+ design that uses MCS locks and Tournament barriers, WiSync delivers an average speedup of 1.12. These are significant improvements. We also see that WiSyncNoT performs on average about the same as WiSync. This is because the wireless Data channel is not very utilized in these applications.

WiSync speeds-up about nine applications. The others have too little synchronization for WiSync to make a difference. WiSync shows its best gains in applications that frequently use barriers, such as *streamcluster* (speedup close to 6) and *ocean*. In addition, significant speedups are also attained in a few lock-intensive applications such as *raytrace* (speedup close to 3) and *radiosity*. Many applications do not use fine-grain synchronization and, therefore, the improvements of WiSync make little difference.

Baseline+ shows low speedups in some applications. This is due to the overhead of its more sophisticated synchronization implementations.

In *dedup* and *fluidanimate*, the locks did not fit in the BM, and we transparently allocated a fraction of the locks in plain memory. However, simulations with an infinitely large BM did not yield any further speedup.

To understand these results better, Table 5 shows the percentage of the cycles in which WiSyncNoT and WiSync use the Data channel. We show data for the most demanding applications and the geometric mean *of all* the applications. From these numbers, we see that, on average, WiSyncNoT and WiSync use the Data channel for 0.2% and 0.1% of the time, respectively. Both utilizations are very low. WiSync's utilization is lower because barrier synchronization uses the Tone channel. In addition, there is little contention. It can be shown that, on average, the latency of a Data channel transfer in WiSyncNoT and WiSync is 9.8 and 5.6 cycles, respectively. Overall, for WiSync to deliver larger speedups, we need applications that use the wireless network more.

7.5 Sensitivity Study

To study the impact of the memory and network latencies on these speedups, we perform a sensitivity study with the configuration variants shown in Table 6. *Default* is the con-

	Str	Rad	W/ns	Flu	Ray	Oc/c	Oc/nc	GM
WT	3.0	2.1	2.0	1.8	1.6	0.8	0.7	0.2
W	0.0	2.1	2.0	1.8	1.6	0.3	0.2	0.1

Table 5: Utilization of the Data channel in WiSyncNoT (*WT*) and WiSync (*W*) in % of the total cycles for the most demanding applications. *GM* is the geometric mean *of all* the applications.

figuration we have used so far. *SlowNet* and *FastNet* increase and decrease, respectively, the network hop latency by two cycles. *SlowNet+L2* additionally makes the L2 cache slower. Finally, *SlowBMEM* makes the BM two cycles slower.

Configuration	L2 RT (Cycles)	BM RT (Cycles)	Net. Hop Lat. (Cycles)
Default	6	2	4
SlowNet	6	2	6
SlowNet+L2	12	2	6
FastNet	6	2	2
SlowBMEM	6	4	4

Table 6: Memory and network configuration variants.

Figure 11 shows the geometric mean speedups of Baseline+, WiSyncNoT, and WiSync over Baseline for the different configurations. The results correspond to 64-core executions. We see that the speedups of WiSync and WiSyncNoT are higher when the on-chip network is slower, and lower when the network is faster. This is because Baseline (and Baseline+) locks and barriers are sensitive to network latency. The impact of the L2 latency is marginal, since all architectures are affected noticeably. Finally, the BM latency barely affects the performance of WiSync and WiSyncNoT, at least for the range considered.

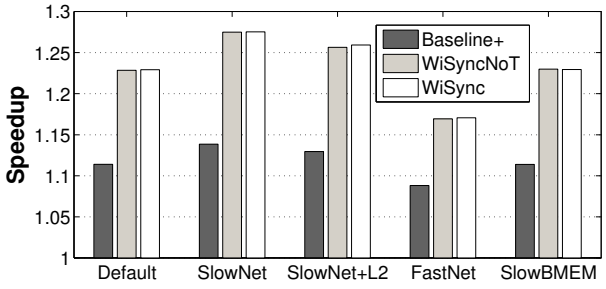


Figure 11: Impact of different memory and network variants on the speedups over Baseline for 64 cores.

8. Related Work

Special hardware support for synchronization. Advanced hardware for synchronization is an active research area (e.g. [5, 7, 12, 15, 25, 26, 37, 38, 40, 53]). Many designs use some form of synchronization in memory or directory, such as the Cray T3E E-registers [38], the SGI Origin fetch-and-Φ operations [25], the buffer of full-empty bit flags in memory banks [53], and the MiSAR on-chip synchronization accelerator [26] to name a few. However, very few designs have ever provided broadcast capabilities, as WiSync does. Perhaps the closest support is Blue Gene/L's network for collectives, broadcast, and reduction [15]. Many designs have optimized barriers, which also include a broadcast step. Some examples are the barrier network in Cray T3D [12], cache coherence protocol optimizations [37], special interconnect hardware for barriers [7], and wired-NOR logic [40].

Transmission Lines (TLs). TLs can provide on-chip broadcast [4, 5, 10, 33, 34, 41, 43]. As a result, they have been used for synchronization. For example, our Tone barrier is inspired in the TLSync TL barrier [33], which also uses tones. Abellan et al. [4, 5] implement locks and barriers using TLs, as they broadcast signals over these TLs. Other designs use TLs for a variety of broadcast networks [10, 34, 43].

Nanophotonics. The transmission of optical signals through nanophotonic waveguides can provide broadcast [21, 23, 46, 47]. Nanophotonics has been proposed for a variety of network architectures [6, 42]. Some proposals use it for on-chip broadcast [21, 23, 46]. Vantrease et al. [47] design a novel form of cache coherence using optical waveguides.

Compared to wireless networks, TLs and nanophotonics are more energy efficient and provide higher bandwidth density. Both advantages are due to the fact that energy is guided rather than radiated. Higher bandwidths can be attained by including several channels within the same link, or by replicating links as in conventional on-chip networks.

On the other hand, network design using TLs and nanophotonics becomes more complex and less scalable than with wireless. They are more complex because both require laying down on the chip a physical infrastructure that interconnects the nodes. TLs are less scalable because they suffer from signal reflections as we connect more nodes to the TL. This requires using amplifying stages between TL segments. Nanophotonics are less scalable due to their laser power needs. Light is modulated by the transmitter and then guided to all the receivers, which extract a fraction of the light each. This causes losses, and requires a high laser power for large destinations sets.

9. Conclusion

This paper proposed to address the challenge of supporting frequent communication due to fine-grain synchronization using on-chip wireless communication. Our architecture, called WiSync, uses a per-core Broadcast Memory (BM). When a core writes to its BM, all the other 100+ BMs get updated in less than 10 processor cycles. In addition to the Data channel for data communication, WiSync proposes the cheap Tone channel, to execute barriers very efficiently. With WiSync, a manycore supports multiprogramming, virtual memory, context switching and, except if the Tone channel is used, process migration. Our evaluation with simulations of 128-threaded kernels and 64-threaded applications showed that WiSync speeds-up synchronization substantially. Compared to using advanced conventional synchronization, WiSync achieved an average speedup of nearly one order of magnitude for the kernels, and 1.12 for the PARSEC and SPLASH-2 applications.

References

[1] S. Abadal, E. Alarcón, M. C. Lemme, M. Nemirovsky, and A. Cabellos-Aparicio. Graphene-enabled Wireless Communication for Massive Multicore Architectures. *IEEE Communications Magazine*, 51(11):137–143, 2013.

[2] S. Abadal, M. Iannazzo, M. Nemirovsky, A. Cabellos-Aparicio, 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*, 23(5):1, 2015.

[3] S. Abadal, B. Sheinman, O. Katz, O. Markish, D. Elad, Y. Fournier, D. Roca, M. Hanzich, G. Houzeaux, M. Nemirovsky, E. Alarcón, and A. Cabellos-Aparicio. Broadcast-Enabled Massive Multicore Architectures: A Wireless RF Approach. *IEEE MICRO*, 35(5):52–61, 2015.

[4] J. L. Abellán, J. Fernández, and M. E. Acacio. GLocks: Efficient Support for Highly-contended Locks in Many-core CMPs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 893–905, May 2011.

[5] J. L. Abellán, J. Fernández, and M. E. Acacio. Efficient Hardware Barrier Synchronization in Many-Core CMPs. *IEEE Transactions on Parallel and Distributed Systems*, 23(8): 1453–1466, 2012.

[6] 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*, 2(2):137–153, 2012.

[7] C. Beckmann and C. Polychronopoulos. Fast Barrier Synchronization Hardware. *Proceedings of Supercomputing*, November 1990.

[8] S. Benedetto and E. Biglieri. *Principles of Digital Transmission with Wireless Applications*. Springer Science and Business Media, 1999.

[9] C. Bienia, S. Kumar, J. 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*, pages 72–81, October 2008.

[10] A. Carpenter, J. Hu, O. Kocabas, M. Huang, and H. Wu. Enhancing Effective Throughput for Transmission Line-based Bus. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 165–176, June 2012.

[11] 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 14th International Symposium on High Performance Computer Architecture*, pages 191–202, February 2008.

[12] Cray Research Inc. CRAY T3D System Architecture Overview, 1993.

[13] S. Deb, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo. Wireless NoC as Interconnection Backbone for Multicore Chips: Promises and Challenges. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2):228–239, 2012.

[14] S. Deb, K. Chang, X. Yu, S. Sah, M. Cosic, P. P. Pande, B. Belzer, and D. Heo. Design of an Energy Efficient CMOS Compatible NoC Architecture with Millimeter-Wave Wireless Interconnects. *IEEE Transactions on Computers*, 62(12): 2382–2396, 2013.

[15] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L System Architecture. In *IBM Journal of Research and Development*, March/May 2005.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann, 2012.

[17] Intel Corporation. Intel Products. ark.intel.com, 2015.

[18] C. H. Jan, M. Agostinelli, H. Deshpande, M. a. El-Tanani, W. Hafez, U. Jalan, L. Janbay, M. Kang, H. Lakdawala, J. Lin, Y. L. Lu, S. Mudanai, J. Park, A. Rahman, J. Rizk, W. K. Shin, K. Soumyanath, H. Tashiro, C. Tsai, P. VanDerVoorn, J. Y. Yeh, and P. Bai. RF CMOS Technology Scaling in High-k/Metal Gate Era for RF SoC (System-on-Chip) Applications. In *Proceedings of the IEEE International Electron Devices Meeting*, pages 604–607, December 2010.

[19] S. Kaya, S. Laha, A. Kodi, D. Ditomaso, D. Matolak, and W. Rayess. On Ultra-short Wireless Interconnects for NoCs and SoCs: Bridging the THz Gap. In *Proceedings of the IEEE 56th International Midwest Symposium on Circuits and Systems*, pages 804–808, August 2013.

[20] B. Khamaisi, S. Jameson, and E. Socher. A 210–227 GHz Transmitter With Integrated On-Chip Antenna in 90 nm CMOS Technology. *IEEE Transactions on Terahertz Science and Technology*, 3(2):141–150, 2013.

[21] 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 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 492–503, December 2006.

[22] T. Krishna, L. Peh, B. Beckmann, and S. K. Reinhardt. Towards the Ideal On-chip Fabric for 1-to-many and Many-to-1 Communication. In *Proceedings of the 44th Annual International Symposium on Microarchitecture*, pages 71–82, December 2011.

[23] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques*, pages 477–488, September 2010.

[24] S. Laha, S. Kaya, D. W. Matolak, W. Rayess, D. DiTomaso, and A. Kodi. A New Frontier in Ultralow Power Wireless Links: Network-on-Chip and Chip-to-Chip Interconnects. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(2):186–198, 2015.

[25] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *International Symposium on Computer Architecture (ISCA)*, June 1997.

[26] C.-K. Liang and Milos Prvulovic. MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 414–426, June 2015.

[27] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, November 1994.

[28] O. Markish, B. Sheinman, O. Katz., D. Corcos, and D. Elad. On-chip mmWave Antennas and Transceivers. In *Proceedings of the 9th IEEE/ACM International Symposium on Networks on Chip*, September 2015.

[29] D. Matolak, A. Kodi, S. Kaya, D. DiTomaso, S. Laha, and W. Rayess. Wireless Networks-on-Chips: Architecture, Wireless Channel, and Devices. *IEEE Wireless Communications*, 19(5), 2012.

[30] F. H. McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Technical report, Lawrence Livermore National Laboratory, 1986.

[31] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[32] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, 1976.

[33] J. Oh, A. Zajic, and M. Prvulovic. TLSync: Support for Multiple Fast Barriers Using On-chip Transmission Lines. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 105–115, June 2011.

[34] 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 22nd International conference on Parallel Architectures and Compilation Techniques*, pages 309–318, September 2013.

[35] J.-D. Park, S. Kang, S. Thyagarajan, E. Alon, and A. Niknejad. A 260 GHz Fully Integrated CMOS Transceiver for Wireless Chip-to-chip Communication. In *Proceedings of the IEEE Symposium on VLSI Circuits*, pages 48–49, June 2012.

[36] T. S. Rappaport, J. N. Murdock, and F. Gutierrez. State of the Art in 60-GHz Integrated Circuits and Systems for Wireless Communications. *Proceedings of the IEEE*, 99(8): 1390–1436, 2011.

[37] J. Sampson, R. González, J. F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder. Exploiting Fine-grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, December 2006.

[38] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[39] E. Seok, D. Shim, C. Mao, R. Han, S. Sankaran, C. Cao, W. Knap, and K. K. O. Progress and Challenges towards Terahertz CMOS Integrated Circuits. *IEEE Journal of Solid-State Circuits*, 45(8):1554–1564, 2010.

[40] S. Shang and K. Hwang. Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):591–605, 1995.

[41] E. Socher and M.-C. F. Chang. Can RF Help CMOS Processors? *IEEE Communications Magazine*, 45(8):104–111, 2007.

[42] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, B. R. Moss, R. Kumar, F. Pavanello, A. H. Atabaki, H. M. Cook, A. J. Ou, J. C. Leu, Y.-H. Chen, K. Asanović, R. J. Ram, M. A. Popović, and V. M. Stojanović. Single-chip Microprocessor that Communicates Directly Using Light. *Nature*, 528(7583):534–538, 2015.

[43] G. Sun, S.-H. Weng, C.-K. Cheng, B. Lin, and L. Zeng. An On-chip Global Broadcast Network Design with Equalized Transmission Lines in the 1024-core Era. In *Proceedings of the International Workshop on System Level Interconnect Prediction*, pages 11–18, June 2012.

[44] R. Ubal, P. Mistry, D. Schaa, H. Ave, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 335–344, September 2012.

[45] M. Uzunkol and G. M. Rebeiz. A Low-Loss 50-70 GHz SPDT Switch in 90 nm CMOS. *IEEE Journal of Solid-State Circuits*, 45(10):2003–2007, 2010.

[46] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. Beausoleil, and J. Ahn. Corona: System Implications of Emerging Nanophotonic Technology. In *Proceedings of the 35th International Symposium on Computer Architecture*, June 2008.

[47] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-free Cache Coherence Protocols. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 132–143, February 2011.

[48] Z. Wang, P. Y. Chiang, P. Nazari, C. C. Wang, Z. Chen, and P. Heydari. A CMOS 210-GHz Fundamental Transceiver with OOK Modulation. *IEEE Journal of Solid-State Circuits*, 49 (3):564–580, 2014.

[49] N. Weissman and E. Socher. 9mW 6Gbps Bi-directional 85–90GHz Transceiver in 65nm CMOS. In *Proceedings of the 9th European Microwave Integrated Circuits Conference*, pages 25–28, October 2014.

[50] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *ACM SIGARCH Computer Architecture News*, 23(2):24–36, 1995.

[51] X. Yu, J. Baylon, P. Wettin, D. Heo, P. Pande, and S. Mirabbasi. Architecture and Design of Multi-Channel Millimeter-Wave Wireless Network-on-Chip. *IEEE Design & Test*, 31(6): 19–28, 2014.

[52] 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*, 62(3):799–806, 2015.

[53] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 35–45, June 2007.