Replica: A Wireless Manycore for Communication-Intensive and Approximate Data

Vimuth Fernando
University of Illinois at Urbana-Champaign
wvf2@illinois.edu

Antonio Franques
University of Illinois at Urbana-Champaign
franque2@illinois.edu

Sergi Abadal
Universitat Politècnica de Catalunya
abadal@ac.upc.edu

Sasa Misailovic
University of Illinois at Urbana-Champaign
misailo@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

Abstract

Data access patterns that involve fine-grained sharing, multicasts, or reductions have proved to be hard to scale in shared-memory platforms. Recently, wireless on-chip communication has been proposed as a solution to this problem, but a previous architecture has used it only to speed-up synchronization. An intriguing question is whether wireless communication can be widely effective for ordinary shared data.

This paper presents *Replica*, a manycore that uses wireless communication for communication-intensive ordinary data. To deliver high performance, Replica supports an adaptive wireless protocol and selective message dropping. We describe the computational patterns that leverage wireless communication, programming techniques to restructure applications, and tools that help with automation. Our results show that wireless communication is effective for ordinary data. For 64 cores, Replica obtains a mean speed-up of 1.76x over a conventional machine. The mean speed-up reaches 1.89x if approximate-computing transformations are enabled. The average energy consumption is substantially reduced by 34% (or 38% with approximate transformations), and the area increases only modestly.

Keywords Approximate; Multicore; Parallelism; Wireless

ACM Reference Format:
Vimuth Fernando, Antonio Franques, Sergi Abadal, Sasa Misailovic, and Josep Torrellas. 2019. Replica: A Wireless Manycore for Communication-Intensive and Approximate Data . In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304033

1 Introduction

Data access patterns where multiple threads interleave reads and writes to the same set of variables in a fine-grained manner and without much per-thread locality do not scale well in shared-memory multiprocessors. They create many network messages, inducing communication bottlenecks. To alleviate this problem, commercial vendors (e.g., [21, 29, 34, 53]) and researchers (e.g., [12, 28, 32, 36, 46, 56, 62, 68]) have proposed various hardware techniques. They include new synchronization and cache coherence protocol improvements, special networks, and new communication technologies such as optics and transmission lines.

Recently, on-chip wireless communication has emerged as a promising alternative that supports fine-grained data sharing with low-latency, and is broadcast-friendly [3, 23, 26, 27]. In this environment, broadcasting a short message of 80 bits takes about 4 ns, which is about two orders of magnitude lower than in conventional on-chip networks. For example, the recent WiSync manycore [3] augments each core with a small antenna and a transceiver. It supports low-latency implementations of synchronization primitives, such as locks and barriers. WiSync stores the state of synchronization variables in a small, per-core Broadcast Memory (BMem) that has identical contents in all of the cores. Writes to the BMem are broadcasted, updating all the BMems at the same time, while reads are satisfied from the local BMem.

While WiSync shows the attractiveness of on-chip wireless communication, it is only tailored to speed-up synchronization operations. An intriguing question is whether the wireless communication and BMem support can be used to speed-up transfers of ordinary data.

Using wireless communication for ordinary data faces two fundamental challenges: the bounded size of BMem and the limited bandwidth of the wireless communication channel. WiSync does not completely experience these challenges, as the synchronization variables typically fit in the 16KB BMem and do not consume much of the wireless channel bandwidth. In contrast, ordinary data does not fit in BMem, and its frequent updates may cause contention in the wireless channel. It is therefore necessary to judiciously select the subset of the data that will benefit the most from the wireless communication, and place it in BMem.

In this paper, we present *Replica*, a manycore architecture and software interface that enables efficient use of wireless communication for ordinary data. We tailor Replica to speed-up *communication-intensive shared data* – whose accesses typically induce substantial overheads in standard cache hierarchies. Our analysis presents several common communication-intensive patterns. They include broadcasts, regular many-to-many interactions, irregular many-to-many interactions, and reductions. To handle these patterns, we present: (i) a software API that exposes BMem to the software developer, and (ii) transformations and tools for selecting communication-intensive data and restructuring applications for improved BMem and wireless channel use.

Further, we propose two hardware-based techniques to reduce contention and latency in the wireless channel. First, we introduce an adaptive wireless protocol. The protocol dynamically identifies whether the data transmissions in the execution are sparse or bursty, and applies a random-access or a token-passing protocol, respectively.

Second, Replica provides hardware support for selectively dropping packets if they carry certain types of data and if the sender encounters a certain level of channel contention. A software developer can use two operations, *approximate locks* and *approximate stores*, to optimize applications that can tolerate noise. Further, she can combine these operations with existing approximation techniques that trade accuracy for reduced communication and/or data size. Together, these techniques have a greater impact on Replica than on standard architectures, due to the limited BMem size and the limited wireless channel bandwidth.

Our results show that Replica effectively uses wireless communication for ordinary data. We evaluated Replica with 10 applications from graph analytics, vision, and numerical simulation. For 64-core executions, Replica speeds-up the applications over a conventional machine by a geometric mean of 1.76x for exact computation and 1.89x for approximate computation. Further, Replica substantially reduces the average energy consumption by 34% (or 38% with approximate computation). Finally, the area increase is small, and the developer effort is modest.

Contributions. Our contributions are: (i) the Replica manycore, with an adaptive wireless protocol and support for selective packet dropping; (ii) software techniques and tools for adapting applications to wireless communication; and (iii) an evaluation of Replica.

2 Background

Figure 1 shows the WiSync architecture [3]. WiSync augments every core of a manycore with a *Broadcast Memory* (BMem), a wireless transceiver, and two antennas (of which we will only consider one). The transceiver has two main modules, namely the physical layer (PHY) and the Medium Access Control (MAC). The PHY module serializes and modulates the data to transmit, detects collisions, and demodulates

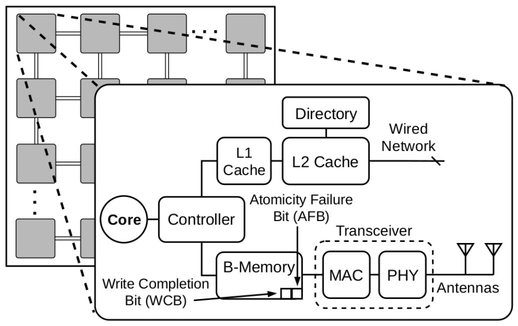

Figure 1. WiSync manycore.

and deserializes data at reception. The MAC module manages the access to the channel by scheduling transmissions and handling collisions [5].

The BMem is a direct-mapped memory of a size similar to an L1 cache. The BMems of all the cores contain the exact same variables that are kept coherent through wireless updates. A core accesses its BMem with plain loads and stores. Based on the physical address of the location accessed, a load or store request is sent either to the L1-L2 hierarchy or to the BMem.

When a core writes to a BMem location, it generates a message to be broadcasted through the wireless network. All BMems (including the local one) are updated simultaneously. This design ensures a total order of writes to BMems across all cores. It also ensures that, at all times, all cores have the same values in their BMems. Loads that access the BMem simply read the local copy of the data.

WiSync uses 5 cycles to transmit a 77-bit packet, which corresponds to a 64-bit write. In the second cycle, the transceiver listens if there was a collision with another packet in the first cycle. If there was no collision, in the next three cycles it sends the rest of the packet with guaranteed no collision. Otherwise, the transfer is aborted, and the senders will retry sending their packets after a randomized, exponentially-increasing number of cycles. This carrier-sensing protocol with exponential backoff adapted to the on-chip scenario is called Broadcast Reliability Sensing (BRS) [38].

WiSync supports read-modify-write instructions that form the basis of synchronization primitives. These instructions leverage a special hardware bit called the Atomicity Failure Bit (AFB). The AFB is set in hardware if, in between the read and write of the local core in a read-modify-write operation to a BMem address, an external core succeeds in performing a write to the same location. In this case, the write of the local core does not occur, and the atomic operation fails.

3 Replica Overview

Replica extends the WiSync architecture in several ways, including the ability to store ordinary (i.e., non-synchronization) *and* synchronization data in the BMem. In this section, we outline the key features of Replica.

Broadcast Memory. Replica provides an API to allocate data in BMem. To store an array `a` in BMem, the developer only needs to change the allocation site to

```
float* a = wireless_malloc(n*sizeof(float));
```

All accesses to the array elements are automatically directed to the BMem, and writes use the wireless channel. Programs do not require any additional developer or compiler interventions, as the BMem is memory mapped. A call to `wireless_free` deallocates the memory.

Since the amount of communication-intensive data may exceed the size of the BMem, it is essential to restructure communication-intensive data structures to fit as much as possible in BMem. We present transformations that allow the flexible storage of a fraction of communication-intensive data in BMem. Our approach rests on two observations: (i) in many applications, the size of communication-intensive data increases at a much slower rate than the full input data size, and (ii) since BMem is memory-mapped, we can transform the data structure layout with little performance penalty.

Adaptive Wireless Protocol. In Replica, the wireless network utilization varies across applications and even within an application. For applications with sparse transmissions, the carrier-sensing protocol from WiSync is sufficient. However, applications with high or bursty load perform better with a token-passing protocol, in which only the node that owns the token can transmit. Replica's MAC module supports both protocols, and switches between the two to adapt to the characteristics of the application. This process is automatic and does not require input from the developer.

Approximate Broadcast Memory. To further reduce the wireless channel contention, Replica uses a section of BMem for approximate data. In this section, the messages for data updates and locking operations may occasionally be dropped, if the latency to perform the access exceeds a certain threshold. Approximate data is allocated as

```
float* a = approx_wireless_malloc(n*sizeof(float));
```

and is stored in a specially-designated section of the BMem. Approximate BMem supports two operations that selectively drop packets:

- **Approximate Store:** It assigns a value `val` to a variable `var` if the write succeeds within a specified latency threshold. Approximate stores can be *unchecked* or *checked*. In the former, if the message is dropped, the computation silently continues without informing the software. In the latter, software can use the call `approx_stac(var, val)` (for store approximate checked) to find out if the write succeeded. Unchecked stores use the same opcode as standard stores. Checked stores use a different opcode.
- **Approximate Lock:** `approx_lock(m)` attempts to obtain the lock `m` within a specified latency threshold. If it succeeds, it returns a success code. If it does not succeed, either because it spins for too long on an already taken lock, or because it takes too long to obtain the wireless network to send the lock acquire update, it returns a failure code. In this case, the software skips the critical section and the unlock operation.

Tool Support. Replica includes a tool infrastructure – a profiler, compiler passes for transformations, and an autotuner – to help the developer restructure the application. Our experience shows that these tools can automate many tasks and enable seamless adaptation of program code to versions of Replica with different hardware characteristics.

4 Replica Architecture

This section describes the two main architectural features of Replica that improve over WiSync – the adaptive wireless protocol and the approximate BMem.

4.1 Adaptive Wireless Protocol

In WiSync, the wireless network is utilized relatively little, and its traffic patterns are simple. Therefore, a wireless MAC protocol like BRS is appropriate. In BRS, when a packet collides, the sender does not try to resend it at the next available opportunity. Instead, it waits for a backoff period before retrying. Specifically, it considers a period of $2^c - 1$ cycles (where c is the number of collisions that the packet has suffered so far), picks a random number within that period, and waits for that number of cycles. The result is a backoff that increases exponentially.

In Replica, the use of the wireless network is more complex, and its utilization patterns vary across applications and within an application. Hence, while Replica retains the BRS protocol for applications or sections of applications with sparse transmissions, it also supports a *Token Ring* protocol in applications with frequent or bursty transmissions [20].

In the Token Ring protocol, there is a logical token that is owned by different nodes at different times. At any time, only the node that owns the token can transmit. At the end of a packet transmission, or if the owner node remains silent for one cycle, the token is passed to the next node following a logical ring.

Replica introduces an adaptive wireless protocol that intelligently switches between the two protocols, adapting to the characteristics of the execution. Specifically, one of the nodes (which we call the *master node*) has a hardware mechanism in its transceiver that monitors the use patterns of the wireless channel and chooses the protocol to use. The mechanism's hardware consists of two counters and simple logic to perform a division and a comparison:

- When running in BRS mode, the counters are called *Coll* and *NoColl*. Every time any core sends a packet, the mechanism checks for a collision. If there is no collision, it increments *NoColl*; otherwise, it increments *Coll*.
- When running in Token Ring mode, the counters are called *Idle* and *Busy*. If the mechanism observes an idle cycle,

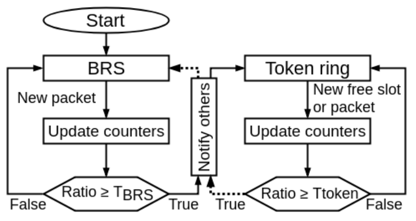

Figure 2. Adaptive wireless protocol.

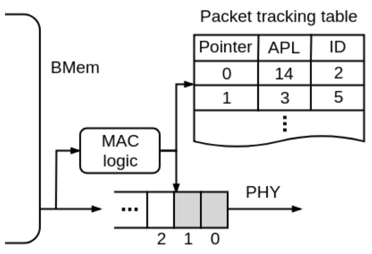

Figure 3. Packet tracking.

```
        Num_spins = 0
        R = TAKEN
Try: exchange R, m_addr
        if (WDB) return -1 //packet dropped
        if (R == TAKEN){  //failure
          if(Num_spins++ > MAX)
            return -1  //too many failures
          jmp Try
        }
        else {    //R is UNTAKEN
          if (AFB){   //atomicity failure
             //no write occurred
             R = TAKEN
             jmp Try
          }
          else return 0  //success
        }
```

Figure 4. Approximate lock acquire.

a core missed its opportunity to transmit, and the *Idle* counter is incremented. Otherwise, a packet is transmitted, and the *Busy* counter is incremented.

Figure 2 presents the operation of the mechanism. The mechanism starts in BRS mode. Following an update to either counter, the hardware calculates the ratio *Coll*/*NoColl*. If this ratio is equal or higher than the T_{BRS} threshold, the transceiver in the master node clears the counters and informs all the nodes to switch to the Token Ring mode.

From then on, the nodes use the Token Ring protocol, and the mechanism in the master node's transceiver computes the ratio *Idle*/*Busy*. When the ratio is equal or higher than the T_{token} threshold, it means that only a few cores have data to transmit and are unnecessarily waiting to get the token. In this case, the transceiver clears its counters and notifies all the nodes to switch to the BRS mode.

4.2 Approximate Broadcast Memory

The programmer can tag a certain range of BMem addresses as containing approximable variables. For variables allocated in this range, Replica can drop wireless packets through approximate stores and locks. This support reduces the pressure when the wireless channel is highly contended.

Approximate Stores. Stores to approximable variables use the wireless channel like any other BMem variable, but the stores can be dropped if the contention for the wireless channel is high. In this case, neither the remote BMems nor the local BMem are updated.

The MAC module keeps track of the waiting delay of droppable write packets. If the waiting delay exceeds threshold T_{drop}, the packet is dropped and the store is canceled. This automatically and dynamically reduces the contention and power in communication-intensive periods. This way, programmers can perform certain approximations only if it is strictly necessary, to avoid saturating the network.

Figure 3 presents the hardware support. It consists of the Packet Tracking Table (PTT) in the MAC module. When a packet is sent to the BMem, it is deposited in a buffer in the transceiver. At this point, if the packet is for a droppable store, it allocates an entry in the PTT. Each PTT entry has three fields. The first one is a pointer to the position of the packet in the buffer. The second one contains the total time that the packet is expected to wait before being sent. We call it the *Accumulated Packet Latency* (APL) for the packet. When the APL for a packet reaches T_{drop}, the packet is dropped. The third field is only used by *checked* approximate stores, and contains the ID of the register that will receive the transmission outcome.

The algorithm to set the APL is protocol dependent. When running in BRS mode, the APL for a packet is set as follows. When the packet arrives at the queue, it sets its APL to the minimum number of cycles that it will have to wait to be sent. This time is equal to the number of cycles remaining in the backoff period of the first packet in the queue, plus the number of packets in the queue times the duration of a packet send (i.e., 5 cycles including the cycle listening for collisions). Note that this computation includes both droppable and non-droppable packets. Further, when a packet is sent and collides with another packet, the hardware calculates the backoff time, and this backoff time is added to the APL of all the droppable packets in the queue (including the packet that collided). Finally, when a packet is dropped, the hardware computes the number of waiting cycles to remove. This number is 5 cycles plus the remaining cycles in the backoff period (if this was the first packet in the queue). This number is subtracted from the APL of all the droppable packets that are queued after the dropped one.

When running in Token Ring mode, the APL for a packet is set as follows. When the packet arrives at the queue, it sets its APL to the number of cycles that it will have to wait assuming an idle wireless channel. This number includes: (i) for the first packet in the queue, the number of cycles remaining until it can start the sending plus the cycles to send the packet, plus (ii) for each of the other packets in the queue, the number of nodes in the machine plus 3 (since a packet takes 4 cycles to be sent now). Further, every time that another node transmits, the hardware adds 3 cycles to the APL of all the entries in the table. Finally, when a packet is dropped, the number of waiting cycles to remove from all the subsequent droppable packets is: (i) if the dropped packet was the first one in the queue, the number of cycles remaining until it could start the sending plus the cycles to send the packet, or (ii) if the packet was not the first one in the queue, the number of nodes in the machine plus 3.

When a packet is dropped, it is removed from the buffer. Further, its PTT entry is removed and, for checked approximate stores, the register indicated in the PTT entry is set. When a droppable packet is sent successfully, the same actions occur, except that the register indicated in the PTT entry is cleared.

Checked Approximate Stores. In unchecked approximate stores, an update may be silently dropped. In the checked version, the programmer obtains the outcome of the store. The programmer uses a special *store approximate* instruction, `sta R1, R2, var_addr`, which takes three arguments: a register `R1` containing the value to store, a register `R2` that indicates if the store was either successfully committed (R2=0) or dropped (R2=1), and the address `var_addr` to receive the data. R2 is the register recorded in the corresponding entry of the PTT. We expose the API call `approx_stac(var, val)`, which is implemented as `sta R1, R2, var_addr; ret R2`. It takes the variable to update (`var`) and the update (`val`), and returns whether the write succeeded (zero) or not (one).

Approximate Locks. To support approximate locks, Replica introduces a new hardware bit in the BMem controller called the *Write Drop Bit* (WDB). The hardware sets the WDB bit when a write packet belonging to a read-modify-write instruction is dropped. The WDB bit remains set until the software reads it, at which point the bit automatically clears. In this way, the programmer is aware of whether an approximate lock has been dropped. Since there is only one read-modify-write instruction executing at a time per core, a single WDB bit is enough.

Figure 4 shows the lock acquire routine for an approximate lock. The code tries to acquire lock `m` in address `m_addr` using an exchange instruction. An approximate lock fails and returns -1 when either (i) the software has unsuccessfully tried to acquire the lock for more than MAX attempts, or (ii) the write packet of the latest read-modify-write instruction that attempted to acquire the lock is queued for T_{drop} cycles. Condition (i) is implemented in software, using variable `Num_spins` (Line 6). When condition (ii) occurs, the write packet is dropped, the WDB bit gets set, and the exchange instruction terminates without performing the write. In this case, the exchange register `R` may have read the new value, but the write to `m_address` has not occurred. Hence, the first action that the software takes after the exchange is to check WDB (Line 4). Irrespective of the current value of `R`, if WDB is set, the function returns -1.

The software then checks `R`. If `R`'s value is still TAKEN and the number of tries is no higher than MAX, the software retries the exchange (Line 8). If `R`'s value is UNTAKEN, we still need to do a final check. WiSync [3] requires the software to check the Atomicity Failure Bit (AFB) (Line 11). If the AFB is set, it means that another node updated `m_address` between the local read and the local write, and that the local write failed. In this case, the software resets `R` to TAKEN and retries the exchange (Line 14).

When the software finds out that an approximate lock operation has failed, it skips the critical section and the subsequent unlock operation. We describe the software transformation in Section 5.3. We have also designed a similar algorithm for compare and swap (CAS) synchronization.

4.3 Other Features

Two additional Replica enhancements over WiSync's wireless hardware are related to the capacity of BMem. First, since the BMem is bigger in Replica than in WiSync, it needs more address bits and is slower. Second, since different applications need different BMem sizes, Replica organizes the BMem in chunks. The chunks that are not allocated by the application are power-gated to save energy.

5 Software Adaptation

In this section, we describe the software infrastructure that we use to leverage the Replica architecture. We start by describing the key access patterns that we target, then discuss our transformations, and finally outline our tool support.

5.1 Communication-Intensive Access Patterns

There are several parallel access patterns that are hard to support in conventional shared-memory multiprocessors. They involve multiple (or all) cores reading from and writing to a particular shared address. They cause communication bottlenecks in current machines. Fortunately, the wireless channel of Replica is especially suited to support these patterns efficiently. We now describe them.

Broadcast. One thread (possibly referred to as the master) writes to a shared address that is subsequently read by many (or all) of the other threads (referred as the workers).

Regular Many-to-Many Interactions. It occurs in codes where different threads operate on sets of overlapping shared addresses, and the communication has regular patterns. Common examples are simulations and numerical applications.

Irregular Many-to-Many Interactions. This pattern is like the previous one except that the inter-thread communication follows irregular patterns. A common example is graph-processing algorithms.

Reduction. Many (or all) threads read and write to a single shared address, aggregating their local contributions.

All these access patterns are easily supported using an address in the BMem. A write by a processor automatically broadcasts the update to all BMems. Since reads are always to the local BMem and writes are observed by all processors quickly, regular and irregular many-to-many interactions are supported trivially. Reductions simply require that processors read and write atomically to the single shared address.

5.2 Transformations to Optimize BMem Utilization

We present several program transformations that enable the BMem to store the most important data, or to store a larger amount of important data.

Data Splitting. This transformation partitions a data structure into important data, which is allocated in BMem, and less important data, which is allocated in regular memory. This allows Replica to deliver high performance even for large data structures that do not completely fit in BMem.

We describe two variants of the transformation. The first variant sets up an ***indirect data structure*** and then partitions the original structure into two parts. For example, consider an array of records. This transformation creates an indirection array with as many pointers as the records, where each pointer points to a record. The latency-critical records are allocated in BMem, while the less important ones in regular memory. All accesses to the original array are then replaced with indirect references. The indirection array (which after the initialization remains read-only) is allocated in the regular memory.

This transformation is flexible, in that we can select any set of fine-grained data to be allocated in the BMem. However, its shortcomings are that we add additional references and that the new array of pointers may evict some data from the regular caches.

The second variant involves ***mapping some of the pages*** of the data structure in the BMem, and mapping the rest in regular memory. For example, we can map the first set of pages of the structure into BMem, or the last set of pages, or an arbitrary set of pages. Compared to the first variant, this approach does not add additional indirections or cause cache evictions. However, it is less flexible, as the grain size of allocation is a page.

In both variants, we redistribute the computation so that all threads participate in processing the data in the BMem.

Data Reduction. These transformations, inspired by other ones from literature, enable BMem to store data more efficiently. As a result, the BMem logically stores a larger amount of important data, potentially reducing program accuracy.

- ***Lock Coarsening*** reduces the number of locks needed to access a given data structure, by making multiple elements of the structure share the same lock [25]. This change reduces the data in the BMem (since only a subset of locks is required) and the inter-core communication, but at the expense of false contention.
- ***Cyclic Collection Update*** sets an upper bound on the memory footprint of a collection, such as a list or a set. If we need to add a new element to the collection that would require an increase in the collection footprint, the new element is dropped or it replaces an existing element. This transformation is inspired by cyclic memory allocation from program repair [43].
- ***Numerical Precision Reduction*** changes the type of the variables stored in the BMem, reducing the size of the variables at the expense of precision. For example, we can change 64-bit `double` types to 32-bit `float` types.

5.3 Transformations to Reduce Communication

Some of these transformations leverage Replica's approximate locks and stores to reduce communication in the wireless channel.

Skipping Critical Sections. We use Replica's approximate locks to occasionally skip critical sections:

```
if (approx_lock(m)==0) { // acquired lock
  // execute critical section
  unlock(m);
}  // else skip
```

In the example, the code tries to acquire the lock. As indicated in Figure 4, if the software unsuccessfully spins for more than a certain number of attempts, or the write packet in a read-modify-write instruction is queued for a certain number of cycles, `approx_lock` returns a non-zero code. In this case, the code skips the critical section. This transformation reduces the communication between cores. It is motivated by a software-only transformation from [7].

Skipping Negligible Updates. This transformation skips updates to a shared variable when the contribution of the update to the value is below a specified threshold. In the following example, the original code (a) adds the variable `upd` to the variable `shared`. In the transformed code (b), if `upd` is smaller than `Threshold`, the update is skipped.

```
upd = local_res();    upd = local_res();
                      if (upd > Threshold){
lock(m);                 lock(m);
shared += upd;           shared += upd;
unlock(m);               unlock(m);}
       (a)                      (b)
```

This transformation reduces the wireless communication if `shared` is allocated in BMem. It is applicable when small updates do not contribute much to the overall solution. However, it changes the computation and its result.

Skipping Updates with Compensation. This transformation skips updates but later tries to compensate for the contribution of the missed updates. For instance, in the following example, variable `var` should receive the sum of all the elements of array `val`. Since the stores use `approx_stac`, they may be dropped. However, if `approx_stac` returns a non-zero status because the contribution of `var[i]` is dropped, subsequent iterations will attempt to add multiple times their contribution to compensate. Finally, if the loop completed without adding the final element(s), they will be aggregated after. In the code, a local variable `fcnt` counts the number of consecutive failed attempts.

```
int fcnt = 0; // failcount
for (i=0;i<MAX;i++){
  if (approx_stac(var, var+val[i]*(1+fcnt))) fcnt++;
  else fcnt = 0;
}
if (fcnt > 0) do {
  lastf = approx_stac(var, var+val[MAX-1]*fcnt);
} while (lastf);
```

This transformation reduces communication in the wireless channel. It relies on the fact that, in many programs, the consecutive updates have similar values. A similar transformation can also be applied to approximate locks.

5.4 Tool Support

To ease program adaptation, we implemented tools that help the developer identify shared data and tune transformations.

Profiler. We developed a memory profiler that detects the shared data in a program. The profiler instruments the memory instructions of the program to record a trace of the memory activity. It then identifies shared variables that are written to by a thread before being read by multiple other threads. It then sorts data structures based on the percentage of addresses that exhibit such patterns, and based on the number of threads that access such addresses. Finally, it presents the report to the developer.

Automated Transformations. We implement the compiler transformations discussed in the previous sections within Clang/LLVM. For instance, for the data splitting transformations in Section 5.2, the developer only needs to write a pragma in the code, and the compiler then generates the code with the structures that best fit in the provided BMem.

Autotuner. The Replica architecture and the program transformations expose parameters that can be tuned to optimize performance. An example of such parameters is the frequency of dropped messages. To explore the space of parameter values and find those that maximize performance subject to accuracy specifications, we develop an autotuner. The autotuner uses the OpenTuner framework [9].

6 Methodology

To evaluate Replica, we perform cycle-level architectural simulations using Multi2sim [60]. We run a variety of applications from SPLASH-2 [63], PARSEC [14], and the CRONO [6] graph suite.

6.1 Applications

Table 1 lists the 10 applications, what they do, and the inputs we use in the evaluation.

Data Sharing Patterns. The benchmark applications have different data-sharing patterns. Water has broadcast communication. The graph applications (BFS, Pagerank, SSSP, CC, and Community) have irregular, mostly many-to-many communication. Volrend mainly contains communication between neighbors, but also has broadcast communication. Canneal has an irregular communication pattern, due to locks. Bodytrack and Streamcluster have one-to-many communications and reductions.

Inputs and Metrics. For the SPLASH-2 and PARSEC applications (except Streamcluster), we use the same input sets as WiSync. For the graph applications, we use input sets from SNAP [2]. The input set sizes were chosen to allow detailed simulation runs that ranged between 4 and 48 hours per run. For the autotuning and profiling runs, we use different, training inputs. These training inputs are as follows. For the graph applications, they are different graphs of the same size and connectivity. For Streamcluster, we generate three new data sets with existing ground truth cluster centers [1]. For the other applications, we use alternative input data sets provided by the application suite.

The last column of Table 1 shows the metrics that we use to compute the accuracy loss of the computations when we use approximation optimizations. We use metrics that have been previously proposed in the literature.

Other Programs. We also analyzed other applications from the SPLASH-2 and PARSEC suites. As noted in previous characterizations [11], most of the remaining programs are data-parallel (e.g., blackscholes and swaptions) or implement regular algorithms with limited sharing, typically among neighbors (e.g., fluidanimate and raytrace). Since we do not expect Replica to improve performance for such computational patterns, we do not evaluate such applications.

6.2 Architecture Configurations

We analyze three configurations of Replica:

- Wireless-Locks (**WL**): it allocates only synchronization variables in BMem. It extends WiSync with the adaptive wireless protocol, and a BMem size that holds all the synchronization variables: 23KB in Water, 39KB in Canneal, and less than 1KB in the rest of the applications.
- Wireless-Optimized (**WO**): it extends WL by allocating some ordinary data in the BMem and applying the Data Splitting and Lock Coarsening transformations (Section 5.2). These transformations preserve the program semantics.
- Wireless-Approximate (**WA**): it extends WO by applying approximation transformations, including Cyclic Collection Update and Numerical Precision Reduction (Section 5.2), the transformations from Section 5.3, and checked and unchecked approximate stores (Section 4.2).

We compare these configurations to a conventional architecture without BMem or wireless network in three configurations: Baseline (**B**) runs the original application, Optimized (**O**) augments B with the transformations in WO, and Approximate (**A**) augments O with the transformations in WA except those that need hardware support (e.g., approximate stores).

Table 2 shows the transformations for each application. The shared variables column lists the non-synchronization

Table 1. Summary of the applications.

Name	Description	Input	Metric
Water [63]	Simulation of water molecules (nsquared)	1000 molecules for 10 steps	Difference in average energies
BFS [6]	Breadth-first search	p2p-gnutella31 (from [2])	Fraction of unvisited nodes
SSSP [6]	Single source shortest path	p2p-gnutella31 (from [2])	Fraction of nodes with different distances
Pagerank [6]	Compute pagerank for nodes in a graph	p2p-gnutella31 (from [2])	Average difference in pagerank
CC [6]	Compute connected components of a graph	p2p-gnutella31 (from [2])	Fraction of nodes with wrong component
Bodytrack [14]	Track a body pose through images	4 frames, 1000 models	Average relative difference of poses
Streamcluster [14]	Cluster streams of points	4096 pts, 20 centers	B^3 clustering metric [8]
Volrend [63]	Render a 3D object	head	Peak Signal to Noise Ratio (PSNR)
Community [6]	Compute modularity of a graph	p2p-gnutella31 (from [2])	Average difference in calculated value
Canneal [14]	Find optimal routing for gates on a chip	10000 elements	Relative difference in routing length

Table 2. Summary of the transformations for different configurations.

Name	Shared Vars (Beyond Synch.)	Optimization (O, WO)	Approximation (A, WA)
Water	`molecules`, `gl_memory`	Data splitting	Precision reduction and skipping critical sections with compensation in function `INTERF`
BFS	`D`	Data splitting	Approximate stores with T_{drop}=75 cycles
SSSP	`D`	Data splitting	Approximate stores with T_{drop}=40 cycles
Pagerank	`PageRank`	Data splitting	Skipping negligible updates with Threshold=0.01
CC	`D`	Data splitting	Approximate stores with T_{drop}=350 cycles
Bodytrack	`mParticles, mWeights, valid`	Command line knob	Approximate stores with T_{drop}=750 cycles
Streamcluster	`feasible`, `work_mem`, `clusterCenters`	Command line knob	Cyclic collection update in function `copycenters`
Volrend	`shading_table`, `out_image`	Data splitting	Approximate stores with T_{drop}=1000 cycles
Community	`comm`	Data splitting	Approximate stores with T_{drop}=2500 cycles
Canneal	`Array of locks`	Lock coarsening	Skipping critical sections in function `swap_locations`

variables stored in the BMem in Replica. The Optimization column presents the semantics-preserving transformations in WO and in O. The Approximation column presents the approximation transformations in WA and, if applicable, in A.

Tuning Approximation Parameters. The Approximation column shows different values of T_{drop} and Threshold (for skipping negligible updates). To select these values for an application, we used the autotuner and executed the application multiple times on a set of different inputs. Our goal was to find the minimum T_{drop} and the maximum Threshold such that the accuracy of the result was acceptable. We present the details in Section 7.5.

6.3 Energy Models

We model the energy consumed by the cores and the memory hierarchy with McPAT [35] and CACTI [42], and the energy of the wired links and routers with DSENT [57]. For the wireless hardware, we compute the power and area consumed per core using data in the literature for 65nm. Specifically, for the transceiver, we use a micrograph and data from [65–67] to estimate an area of 0.25mm^2 (including passives) and a power of 30mW. For the data converter, based on [64], we estimate an area of 0.03mm^2 and a power of 0.72mW. For the serializer and deserializer, data from [52] indicates an area of 0.04mm^2 and a power of 10.8mW. Finally, for the antenna, [30] shows a simple dipole that consumes an area of 0.04mm^2. We double the area to 0.08mm^2 to make it more realistic. Therefore, the overall RF circuit with passives consumes 0.4mm^2 and 41.5mW.

However, following [41, 65], we can power gate the transmitter's power amplifier and the receiver's low noise amplifier when not in use. This saves 10mW for the transmitter and 10mW of the receiver. The remaining components (serializers, data converters, oscillators, mixers, and detectors) are always on. Moreover, since our data comes from 16Gb/s systems and we use a system with 20Gb/s, we need to scale up the power consumption linearly. This gives us the following total values for the per-core RF circuitry at 65nm: 39.4mW when the transmitter is idle, 39.4mW when the receiver is idle, 26.9 mW when both are idle, and a total area of 0.4mm^2.

The next step would be to scale these numbers to the 22nm technology assumed for the core. Several authors [4, 19] argue that the area reduces linearly with the feature size. However, we conservatively use no scaling, and keep the area at 0.4mm^2 and the power at 39.4mW with gating. Note that these numbers are much higher than those used in WiSync, which are 0.14mm^2 for the area and 18mW for the power. Finally, using the amplifier consumption of [65], and the power-gating overheads of [41], we estimate a transient energy of 1.14pJ.

6.4 Simulator Implementation

We use cycle-level execution-driven simulations using the Multi2sim [60] simulator. We model a manycore with 32–64 cores at 22nm technology running at 1GHz. Table 3 shows the parameters of the architecture. Each tile has a 2-issue out-of-order core, 32KB of private L1 instruction and data caches, and a 512KB bank of shared L2. The NoC is a 2D mesh. The per-core BMem is as large as an L2 bank, but we power-gate unused 32KB chunks as directed by the application. We will present the used fraction of BMem in the next section. The wireless network has a data rate of 20 Gb/s, enough to transmit a BMem line and its address (about 80 bits) in 4 cycles (plus one cycle for collision detection). We do not consider missing packets due to noise, since the error rate is below 10^{-16}. We augment Multi2sim with an on-chip wireless network that accurately models transmissions, collision handling, transceiver power-gating, and packet dropping.

Table 3. Architecture modeled. RT means round trip.

General Parameters	
Architecture	Manycore with 32–64 cores at 22nm technology
Core	Out of order, 2-issue wide, 1GHz, x86 ISA
ROB; ld/st queue	64 entries; 20 entries
L1 I+D caches	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 (default), 2 or 1 cycles/hop, 128-bit links
Off-chip memory	Connected to 4 mem controllers, 110-cycle RT
Replica Parameters	
Per-core BMem	Up to 512KB, in 32KB chunks (Table 5) 6-cycle RT, 64-bit wide line
Wireless channel	20Gb/s; 1 cycle for collision detection
MAC Protocols	BRS (exponential backoff), token passing in ring
MAC Thresholds	$T_{BRS} = 0.4$, $T_{token} = 15$
T_{drop}	40–2500 cycles (Table 2)
Transceiv+Anten	Area: 0.4mm^2; TX/RX/idle: 39.4/39.4/26.9mW
Power gating	Analog amplif. (transient: 1.14 pJ), unused BMem

7 Evaluation

In our evaluation, we examine the performance of Replica (Section 7.1), the effect of the adaptive wireless protocol (Section 7.2), the energy consumption and area of Replica (Section 7.3), the impact of approximations on accuracy (Section 7.4), the relationship between tuning parameters and the accuracy (Section 7.5), how applications are adapted for Replica (Section 7.6), and a sensitivity analysis of architectural parameters (Section 7.7).

7.1 Analysis of Performance

Figures 5 and 6 present the speedup of the different architecture configurations over Baseline (B) for 64 and 32 core architectures, respectively. The X-axis of the plots lists the configurations: **B**aseline, **O**ptimized, **A**pproximate (when applicable), **W**ireless-**L**ocks, **W**ireless-**O**ptimized, and **W**ireless-**A**pproximate. The Y-axis is the speedup, computed as the ratio of the execution times of the B configuration and the other configuration.

From Baseline (B) to Baseline Replica (WL). The difference between these two bars is the effect of using wireless communication for synchronization variables and the support for the adaptive wireless protocol. From the figure, we see that the average speedup of WL is 1.40x for 64 cores and 1.13x for 32 cores. For the applications in common with the WiSync paper, the numbers are largely similar, except for Streamcluster, which uses a different input set. We discuss the impact of the adaptive wireless protocol in Section 7.2.

From Baseline Replica (WL) to Optimized Replica (WO). We now consider the impact of the exact transformations. Since such transformations have virtually no impact on the baseline architecture (i.e., the difference between O and B is minimal), we focus only on the wireless configurations.

WO improves performance over WL for all the applications. On average, these improvements translate into an average speedup of 1.27x for 64 cores and 1.30x speedup for 32 cores. This shows the benefits of wireless transfers of optimized ordinary data. BFS and Streamcluster are communication heavy and thus benefit the most from these transformations. Most of the other applications have large improvements as well. Even Volrend, the application with the smallest gains, still manages to obtain speedups of about 10%.

From Optimized Replica (WO) to Approximate Optimized Replica (WA). Allowing approximations further increases the speedups in six applications, while in four applications there is practically no change. The average speedup of WA over WO is 1.08x for 64 cores and 1.06x for 32 cores (but can go up to 1.27x for CC on 32 cores). In most of the graph applications, Volrend, and Bodytrack, the use of approximate stores reduces the contention on the wireless network, and makes the remaining communications faster. In Water, the majority of savings come from precision reduction, as it allows storing more molecules in the BMem. Since the computation has frequent broadcasts, the overall speed of data transfers is improved. Approximations do not improve Canneal or Streamcluster: in Canneal, the contention for locks is small, while in Streamcluster the provided input does not benefit from the approximation.

In the architecture without wireless network, these transformations are only applicable to three applications. Further, even in these applications, the impact is generally smaller than in Replica. This is because the reduction of load in the network is more beneficial in the bandwidth-limited wireless network. Overall, on average across all applications, the difference between A and O is minimal.

Summary of speedups. Overall, the speedup of the exact version of Replica (WO) over the optimized baseline (O) is 1.76x. If we add the approximations, the speedup of the approximate Replica (WA) over the approximate baseline (A) is 1.89x.

Effect of Individual Approximation Transformations. Comparing the WA and WO bars in Figures 5 and 6, we see the impact of individual transformations listed in Table 2. For example, dropping negligible updates in Pagerank generates a 1.14x speedup for 64 cores and 1.05x for 32 cores. Skipping critical sections in Water (only one of its transformations) and Canneal can be shown to generate, on average, only a 1.02x speedup for 64 cores and 1.01x for 32 cores. The executions of these two benchmarks skipped 0.75% of all critical sections. These speedups are modest because these two applications have low lock contention.

The approximate stores optimization applied to six applications in Table 2 can be shown to generate, on average, a 1.08x speedup for 64 cores and 1.06x for 32 cores. To estimate the effect of this optimization on an architecture without wireless network, we modified the applications to skip writes to the same data structures at the same frequency as write packets were dropped in the wireless network. We then run the applications on the O architecture, and obtain a lower average speedup of 1.04x for 64 cores.

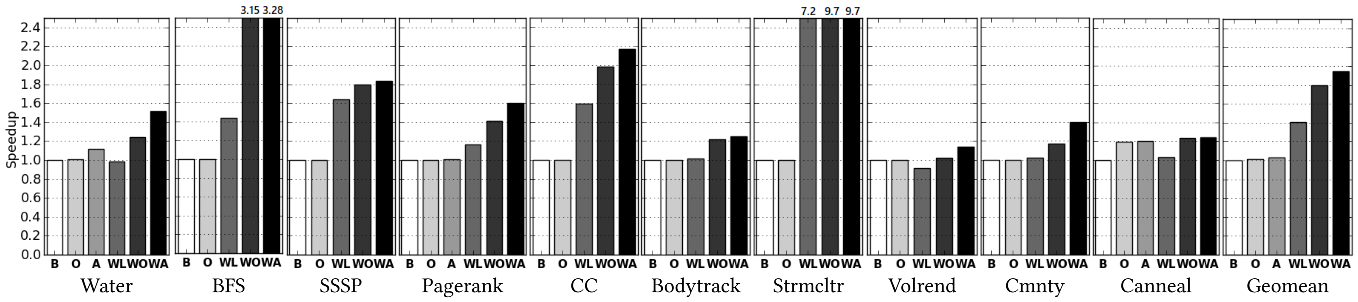

Figure 5. Speedups of the different configurations over Baseline (B) for 64 cores.

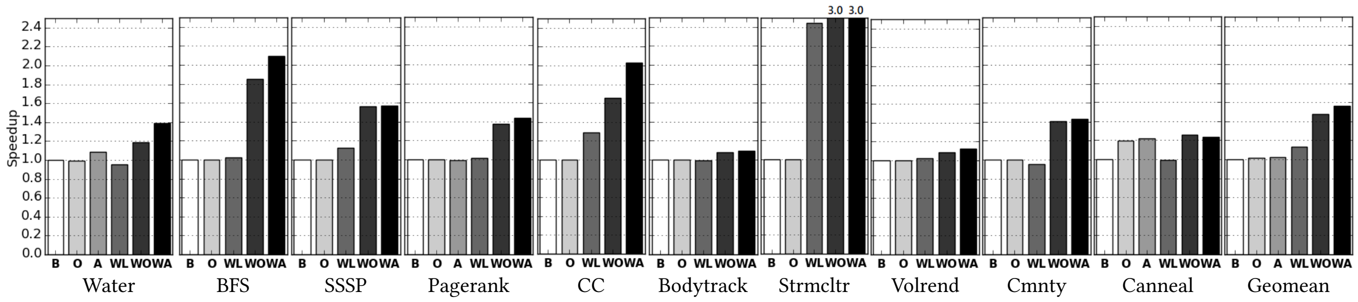

Figure 6. Speedups of the different configurations over Baseline (B) for 32 cores.

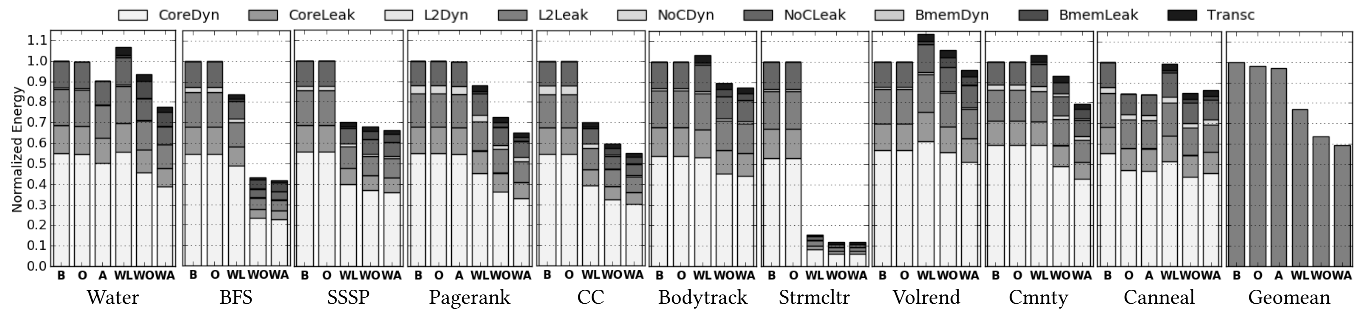

Figure 7. Energy consumed by the different configurations relative to Baseline (B) for 64 cores.

Synergy of Approximate Transformations. Finally, we spotlight an additional optimization opportunity to combine transformations that skip communication with those that skip computation. We have implemented an additional optimization in Pagerank, where if an update to the page rank of a element is below a threshold in one iteration, the program skips updating the page rank in all subsequent iterations, thus reducing computation. The speedup of WA over WO obtained with this transformation with 64 cores is 2.10x. This is higher than the 1.60x speedup of A over O attained in an architecture without wireless network, for the same accuracy.

7.2 Adaptive Wireless Protocol

In our experiments, we run the adaptive mechanism of Figure 2 for an initial section of the execution of the application, until the hardware can identify which of the two protocols is best for the application. Specifically, the execution of an application is logically divided into intervals of 10,000 cycles. At the beginning of each interval, the two counters discussed in Section 4.1 start at zero. Then, as execution proceeds, they get updated. At the end of the interval, based on their relative values and the values of the T_{BRS} and T_{token} thresholds (Table 3), the hardware decides what protocol to try in the next interval, and clears the counters. The process then starts again. After about 350 intervals on average, the hardware makes the decision to stick with the protocol that has been chosen in most of the intervals so far.

Of our applications running under WO, seven end-up sticking with the token ring protocol (BFS, SSSP, Pagerank, CC, Streamcluster, Volrend, and Community), and three with the BRS protocol (Water, Bodytrack, and Canneal). In most applications, the percentage of intervals when the dominant protocol was chosen is greater than 75%. The percentages are lower in BFS (60%), Pagerank (51%), and Volrend (51%).

To assess the performance impact of our adaptive wireless protocol, we measure how the execution time of WO in Figure 5 would change if either all applications were using BRS or all were using token ring. Specifically, if the applications

Table 4. Output accuracy.

Benchmark	A	WA
Water	0.083	0.0004
BFS	-	0.0002
SSSP	-	0.046
Pagerank	0.024	0.024
CC	-	0.0007
Bodytrack	-	0.099
Streamcluster	-	1.00
Volrend	-	37.2 dB
Community	-	0.07
Canneal	0.0001	0.0004

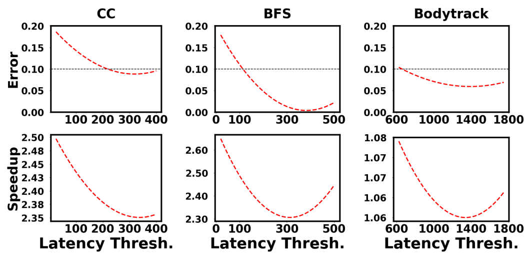

Figure 8. Autotuning latency thresholds (T_{drop}).

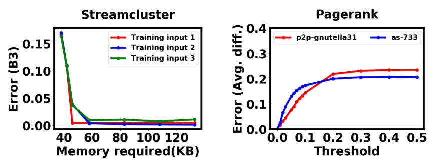

Figure 9. Tunable accuracy profiles.

that prefer token used BRS, their individual execution times would be higher by 28.1% (Community), by 27.1% (CC), by 21.7% (Pagerank), and by less than 2.2% (BFS, SSSP, Streamcluster, and Volrend). Conversely, if the applications that prefer BRS used token, their individual execution times would be higher by 8.6% (Bodytrack) and by less than 2.0% (Water and Canneal). With these results, we can conclude that, if all the applications used BRS, the average execution time under WO would increase by 8.4%; if all the applications used token ring, the average execution time would increase by 1.2%.

7.3 Analysis of Energy and Area

Energy Impact. Figure 7 shows the energy consumed by the different configurations relative to B for 64 cores. The figure is organized as Figure 5. The energy is broken down into dynamic (*Dyn*) and leakage (*Leak*) energy for the core (including L1), L2, NoC, and BMem. We also show the contribution of the transceiver.

The results show that about half of the energy is dynamic energy in the core, and the rest is leakage energy in all the components. Across the bars, we see that the energy savings of each configuration over the Baseline (B) are broadly proportional to the performance improvements of the configuration over B. The wireless configurations reduce the cost of the polling operations and long distance communications. Therefore, they reduce the energy consumption. This effect is especially visible in Streamcluster.

The energy cost of the wireless communication itself can be estimated by adding the contributions of the BMem and the transceiver. We see that such contribution is modest. On average, it is 8.9% of the energy in WO and 8.6% of the energy in WA. The results for 32 cores show a similar behavior and are omitted to save space.

Summary of energy savings. Overall, the average energy savings of the exact version of Replica (WO) over the optimized baseline (O) are 34%. The average energy savings of the approximate Replica (WA) over the approximate baseline (A) are 38%.

Area Impact. Based on the numbers from Section 6.3 and Table 3, our tools estimate that the area overhead of supporting wireless communication is around 15.5% of the Replica architecture. This includes the contribution of the BMems (11.52%) and of the transceivers and antennas (3.97%). Note that these results are conservative, as we do not scale the area of the RF components from 65nm to 22nm (Section 6.3).

7.4 Analysis of Accuracy

Table 4 presents the accuracy losses of the A and WA executions analyzed in Section 7.1 for 64 cores. The accuracy losses are based on the accuracy metrics defined in Table 1. Configuration A does not exist in some applications because the optimizations applied are not supported in conventional architectures (e.g., the approximate stores) or are not useful (e.g., the cyclic collection updates). It can be shown from past literature [16, 37, 54] that the levels of accuracy loss presented are considered acceptable.

Six applications use approximate stores. On average, 4% of all stores were dropped in our applications. The graph benchmarks implement iterative algorithms where each iteration improves on the results. Approximate stores may cause skipping an update to a node's value. However, the computation for that node will be redone in a future iteration, reducing the final error. In Volrend, approximate stores cause a small effect on the PSNR of the output image. Our inspection shows that only 0.8% of the pixels differ by 10% or more from the Baseline (7% of the pixels differ by more than 5%). In Bodytrack, approximate stores cause the model calculations to be done on stale data. Because Bodytrack aggregates a large number of models, errors in a few models have a small impact on accuracy.

The approximate version of Pagerank skips updates to the shared state if the value is below a given threshold. With a threshold value of 0.01, the approximate version produces the same top 10 and top 100 elements.

Water is a simulation that can typically handle the small loss of precision from double to float conversion. While skipping updates can cause errors to amplify across time steps, using the compensation significantly reduces this effect. In Canneal, lock coarsening and skipping critical sections cause minimal changes in the generated netlist.

In Streamcluster, the approximation overwrites cluster centers if the allocated list of centers is already full. For the provided input, all intermediate centers fit into the list without the approximation. Section 7.5 shows the impact on accuracy for larger inputs.

7.5 Profiles of Tunable Approximations

We study the relationship between the accuracy of the computation, the approximation parameters, and different inputs.

Approximate Stores. Since the latency threshold for dropping packets in approximate stores (T_{drop}) needs to be specified by the software, we used our autotuner to identify good T_{drop} values. Our goal is to attain a given level of accuracy (i.e., more than 40 dB for Volrend and less than 10% error for other applications).

Figure 8 shows the results of autotuning experiments for CC, BFS, and Bodytrack. The figure shows how the error and speedup change with T_{drop} values. Typically, as the autotuner reduces T_{drop}, the application accuracy changes a little, until the point where the error rate dramatically increases. The remaining applications exhibit similar behavior. Using these models, the autotuner selected the T_{drop} values that we used with the production inputs of Table 1.

Streamcluster. The number of cluster centers controls the size of the memory allocated in the BMem. The Streamcluster input provided by PARSEC consists of uniformly-distributed centers, and is not suitable for accuracy analysis. We therefore used alternative inputs (Section 6.1). Figure 9 presents the accuracy as a function of the size of the data structure that contains the intermediate cluster centers. Each line is generated by a different training input. At 60 KB, the number of intermediate centers is around 200, which is more than enough to contain all the actual cluster centers. We see that, for 60 KB or higher, the error is negligible.

Pagerank. In Pagerank, we conditionally skip updates if they are below a certain threshold. We analyze the impact of using different thresholds on accuracy for two inputs. Figure 9 presents the results for inputs p2p-gnutella31 and as-733 (from [2]). For both inputs, when the threshold values are small (i.e., fewer updates are dropped), the error is low. As the threshold increases, more messages are dropped and the error increases.

7.6 Adapting Applications to Replica

Table 5 shows how we adapt applications for Replica. It shows the lines of code in the program (Column 2), the number of lines affected by Replica's transformations (Column 3), the size of the data we place in BMem (Column 4), the fraction of application's data in BMem (Column 5), and the number of data structures allocated in BMem vs. the number of structures that the profiler identified as shared among all threads, including synchronization data structures (Column 6).

The results show that the changes to the code are typically small. Moreover, the fraction of the application's data that is placed in BMem is typically very small – only Water and SSSP are exceptions. Also, the size of such data is typically only 100–300KB. In each application, we power-up as many 32KB chunks of BMem as needed to hold this data.

Table 5. Statistics on how programs are adapted for Replica.

Name	LOC	Affected Lines	Data in BMem	% Data in BMem	Allocated vs. Profiled
Water	1641	10	352 KB	26.0%	2 vs. 2
BFS	475	10	245 KB	0.0%	2 vs. 2
SSSP	351	30	245 KB	23.2%	2 vs. 2
Pagerank	375	20	66 KB	0.8%	2 vs. 2
CC	557	10	245 KB	1.4%	2 vs. 2
Bodytrack	8672	24	121 KB	8.7%	n/a
Streamcluster	1660	8	137 KB	16.4%	3 vs. 4
Volrend	2604	4	147 KB	0.6%	3 vs. 3
Community	580	15	245 KB	0.0%	2 vs. 2
Canneal	2886	50	39 KB	0.1%	2 vs. 2

Table 6. Speedups for different cycles per hop (C/H) in the wired network.

Speedup	**64 cores**			**32 cores**		
Metric	C/H=4	C/H=2	C/H=1	C/H=4	C/H=2	C/H=1
A/WA	1.89	1.51	1.41	1.52	1.37	1.32
O/WO	1.76	1.39	1.31	1.45	1.29	1.23
WL/WO	1.27	1.12	1.12	1.30	1.17	1.17
WO/WA	1.08	1.09	1.08	1.06	1.06	1.07

Profiler. In all applications except Bodytrack, the profiler identified all the data structures shared by all the threads. This includes synchronization data structures, such as barriers. We allocated these in the BMem. In Bodytrack, the profiler could not instrument the C++ `std::vector` allocator.

Data Scaling. We also studied how the size of the data that we want to place in BMem scales with input data size. For the graph applications, such data consists of nodes with many neighbors. We studied 20 graphs with 100K–3M nodes from the popular SNAP dataset of graphs [2]. In 16 of these graphs, all nodes with high sharing (at least 8 neighbors) do fit inside the BMem for our applications. Even some graphs of size 10M nodes will fit, if we limit the storage in BMem to nodes with at least 32 neighbors.

For the other applications, the size of the data that we want to place in BMem scales as follows. For Water, it scales linearly with the number of molecules, but independently of the number of steps; for Bodytrack, with the number of models used, but independently of the size or number of frames; for Volrend, with the size of the image, but independently of the number of rendering steps; for Canneal, with the number of locks, but independently of the number of circuit gates; and for Streamcluster, with the number of intermediate centers, but independently of the total number of data points.

7.7 Sensitivity to Architectural Parameters

Latency of the Wired Network. Our default wired NoC has a latency of 4 cycles per hop (Table 3). In this section, we re-evaluate Replica with wired NoCs that have a latency of 2 or 1 cycles per hop. Table 6 shows the resulting values of various speedups for different cycles per hop and different core counts. Each number is the geometric mean of all the applications. The table shows that, as the wired network becomes faster, the Replica speedups (A/WA, O/WO, and WL/WO) decrease. However, even for the fastest, 1-cycle per hop NoC, the speedups are considerable. The speedups due to approximations (WO/WA) remain unchanged.

Bigger L2 Cache. We have increased the size of the L2s of the Baseline (B) architecture from 512KB to 1MB per core, to use the same storage as a worst-case Replica – although Replica only uses a fraction of its BMem (Table 5). We find that this change only speeds-up Baseline by 1.04x for 64 cores.

8 Related Work

Wireless Architectures. We described WiSync [3] in Section 2. Duraisamy et al. [27] accelerate graph analytics using an NoC augmented with wireless links to better support irregular communication patterns. In their case, the application is oblivious of the underlying architecture, and the routing mechanism of each node decides whether to use the wireless links or the regular wire lines, based on the destination address. Their work is also different from ours in that the wireless links are only used to unicast packets between distant cores, irrespective of their criticality, and just as a way to shorten the propagation time of the packets through the network. Later, Duraisamy et al. [26] propose to accelerate graph analytics by bypassing certain updates. Their approximation is exclusively software-based and reduces both the computation and the volume of data lookups, specific to a particular community detection graph algorithm. In contrast, Replica presents hardware-supported, general approximate store and approximate lock mechanisms, which we applied across multiple application domains.

Nanophotonics and Transmission Lines. Transmission of optical signals through nanophotonic waveguides [12, 32, 33, 58, 62] and transmission of radiofrequency signals through transmission lines (TLs) [13, 17, 18, 45, 55, 59] can provide broadcast. Compared to wireless networks, both nanophotonics and TLs are more energy efficient and provide higher bandwidth, because energy is guided rather than radiated. However, network design using either nanophotonics or TLs becomes more complex and less scalable than wireless. It is more complex because it requires a physical infrastructure that interconnects the nodes. Nanophotonics are less scalable due to laser power needs. Light is modulated by the transmitter and then guided to all the receivers. Each receiver extracts a fraction of the light, causing losses, and requiring high laser power for large destinations sets. TLs are less scalable due to: (1) the need to overcome signal reflections with amplifying stages between segments, which are costly and complicate the design, (2) the requirement of a centralized arbiter for the bus, (3) the fact that the analog logic in TLs cannot handle broadcast operations well, especially if the fan-out is large.

Scratchpads. While both BMem and scratchpads [10] have a finite size, BMems are automatically coherent. They do not rely on the compiler to keep them coherent, which is a major reason for the difficulty of using scratchpads. In Replica, the programmer and/or compiler just allocates the data in BMem and Replica transparently handles coherence in hardware.

Lossy NoCs. Prior work has proposed to apply lossy compression techniques to messages before sending them to the network [15]. The approximation occurs in the (wired) network interface, but could be potentially applied to wireless too. Although bufferless networks [22, 44] drop or deflect packets to undesired paths when there is contention at the switches, they are not approximate, since delivery is ensured through retransmissions.

Approximate Parallelization. Relaxed synchronization optimizations intentionally give up some synchronization for faster execution (e.g., [16, 24, 31, 37, 39, 40, 47–51, 61]). The previous works mainly show the potential of many computations to successfully continue execution with relaxed synchronization and random errors on commodity hardware. Our paper presents an approximate BMem architectural abstraction that is specialized for packet dropping. We show the efficiency of our hardware and software co-design and develop a toolchain to automate program adaptation.

9 Conclusion

This paper presented Replica, a manycore that uses wireless communication for communication-intensive ordinary data. Replica supports two hardware mechanisms to reduce contention and latency in the wireless channel: an adaptive wireless protocol and the ability to selectively drop wireless packets if the sender encounters a certain level of contention. We also described the computational patterns that can leverage wireless communication, and exact and approximate programming techniques to restructure applications.

Our results showed that Replica effectively uses wireless communication for ordinary data. For 64-core executions, Replica sped-up applications over a conventional machine by a geometric mean of 1.76x for exact computation and 1.89x for approximate computation. Further, Replica substantially reduced the average energy consumption by 34% (or 38% with approximate computation). Finally, the area increase is small, and the developer effort modest.

Acknowledgments

We would like to thank Keyur Joshi for his help with compiler transformations. This work was funded in part by NSF Grants No. CCF-1629431 and CCF-1703637.

References

[1] 2018. Sci-Kit Learn. `scikit-learn.org`.
[2] 2018. Stanford Network Analysis Project `snap.stanford.edu`.
[3] Sergi Abadal, Eduard Alarcón, Albert Cabellos-Aparicio, and Josep Torrellas. 2016. WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication. In *ASPLOS*.

[4] Sergi Abadal, Mario Iannazzo, Mario Nemirovsky, Albert Cabellos-Aparicio, Heekwan Lee, and Eduard Alarcón. 2015. 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 (2015).

[5] Sergi Abadal, Albert Mestres, Josep Torrellas, Eduard Alarcón, and Albert Cabellos-Aparicio. 2018. Medium Access Control in Wireless Network-on-Chip: A Context Analysis. *IEEE Communications Magazine* 56, 6 (2018).

[6] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *IISWC*.

[7] Riad Akram, Mohammad Mejbah Ul Alam, and Abdullah Muzahid. 2016. Approximate Lock: Trading off Accuracy for Performance by Skipping Critical Sections. In *ISSRE*.

[8] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. 2009. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information retrieval* 12, 4 (2009).

[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *PACT*.

[10] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *CODES*.

[11] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of SPLASH-2 and PARSEC. In *IISWC*.

[12] Christopher Batten, Ajay Joshi, Vladimir Stojanovic, and Krste Asanovic. 2012. Designing Chip-Level Nanophotonic Interconnection Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (2012).

[13] Bradford M. Beckmann and David A. Wood. 2003. TLC: Transmission Line Caches. In *MICRO*.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*.

[15] Rahul Boyapati, Jiayi Huang, Pritam Majumder, Ki Hwan Yum, and Eun Jung Kim. 2017. APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures. In *ISCA*.

[16] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing program semantics to unleash parallelization. In *CGO*.

[17] Aaron Carpenter, Jianyun Hu, Ovunc Kocabas, Michael Huang, and Hui Wu. 2012. Enhancing effective throughput for transmission line-based bus. In *ISCA*.

[18] Aaron Carpenter, Jianyun Hu, Jie Xu, Michael Huang, and Hui Wu. 2011. A case for globally shared-medium on-chip interconnect. In *ISCA*.

[19] M Frank Chang, Jason Cong, Adam Kaplan, Mishali Naik, Glenn Reinman, Eran Socher, and Sai-Wang Tam. 2008. CMP Network-on-Chip Overlaid With Multi-Band RF-Interconnect. In *HPCA*.

[20] D.D. Clark, K.T. Pogran, and D.P. Reed. 1978. An introduction to local area networks. *Proc. IEEE* 66, 11 (1978).

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

[22] Bhavya K Daya, Li-shiuan Peh, and Anantha P Chandrakasan. 2016. Quest for High-Performance Bufferless NoCs with Single-Cycle Express Paths and Self-Learning Throttling. In *DAC*.

[23] Sujay Deb, Amlan Ganguly, Partha Pratim Pande, Benjamin Belzer, and Deukhyoun Heo. 2012. Wireless NoC as Interconnection Backbone for Multicore Chips: Promises and Challenges. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (2012).

[24] Enrico A Deiana, Vincent St-Amour, Peter A Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. In *ASPLOS*.

[25] Pedro C Diniz and Martin C Rinard. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel and Distrib. Comput.* 49, 2 (1998).

[26] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Aananth Kalyanaraman. 2017. Accelerating Graph Community Detection with Approximate Updates via an Energy-Efficient NoC. In *DAC*.

[27] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Ananth Kalyanaraman. 2016. High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics. *ACM Trans. Embed. Comput. Syst* 15, 26 (2016).

[28] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. 2015. Coherence Domain Restriction on Large Scale Systems. In *MICRO*.

[29] 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. 2005. Overview of the Blue Gene/L System Architecture. In *IBM Journal of Research and Development*.

[30] Felix Gutierrez, Shatam Agarwal, Kristen Parrish, and Theodore S. Rappaport. 2009. On-chip integrated antenna structures in CMOS for 60 GHz WPAN systems. *IEEE Journal on Selected Areas in Communications* 27, 8 (2009).

[31] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu. 2018. On Approximate Speculative Lock Elision. *IEEE Transactions on Multi-Scale Computing Systems* 4, 2 (2018).

[32] N. Kirman, M. Kirman, R. K. Dokania, Jose F. Martinez, Alyssa B. Apsel, Matthew A. Watkins, and David H. Albonesi. 2006. Leveraging Optical Technology in Future Bus-based Chip Multiprocessors. In *MICRO*.

[33] George Kurian, J.E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, L.C. Kimerling, and Anant Agarwal. 2010. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *PACT*.

[34] J. Laudon and D. Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*.

[35] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*.

[36] Ching-Kai Liang and Milos Prvulovic. 2015. MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management. In *ISCA*.

[37] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. 2009. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS*.

[38] Albert Mestres, Sergi Abadal, Josep Torrellas, Eduard Alarcón, and Albert Cabellos-Aparicio. 2016. A MAC protocol for Reliable Broadcast Communications in Wireless Network-on-Chip. In *Proceedings of the 9th International Workshop on Network on Chip Architectures*.

[39] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 88.

[40] Sasa Misailovic, Stelios Sidiroglou, and Martin C Rinard. 2012. Dancing with uncertainty. In *RACES*.

[41] Hemanta Kumar Mondal, Shashwat Kaushik, Sri Harsha Gade, and Sujay Deb. 2017. Energy-Efficient Transceiver for Wireless NoC. In *VLSID*.

[42] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report.

[43] Huu Hai Nguyen and Martin Rinard. 2007. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ISMM*.

[44] GP Nychis, Chris Fallin, and Thomas Moscibroda. 2012. On-chip networks from a networking perspective: congestion and scalability in many-core interconnects. In *SIGCOMM*.

[45] Jungju Oh, Milos Prvulovic, and Alenka Zajic. 2011. TLSync: support for multiple fast barriers using on-chip transmission lines. In *ISCA*.

[46] Jungju Oh, Alenka Zajic, and Milos Prvulovic. 2013. Traffic Steering Between a Low-latency Unswitched TL Ring and a High-throughput Switched On-chip Interconnect. In *PACT*.

[47] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with relaxed synchronization. In *Relax*.

[48] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *SC*.

[49] Martin Rinard. 2013. Parallel Synchronization-Free Approximate Data Structure Construction. In *HotPar*.

[50] Martin C Rinard. 2007. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*.

[51] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: pattern-based approximation for data parallel applications. In *ASPLOS*.

[52] Saurabh Saxena, Guanghua Shu, Romesh Kumar Nandwana, Mrunmay Talegaonkar, Ahmed Elkholy, Tejasvi Anand, Woo Seok Choi, and Pavan Kumar Hanumolu. 2017. A 2.8 mW/Gb/s, 14 Gb/s Serial Link Transceiver. *IEEE Journal of Solid-State Circuits* 52, 5 (2017).

[53] S. Scott. 1996. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS*.

[54] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*.

[55] Eran Socher and Mau-Chung Frank Chang. 2007. Can RF Help CMOS Processors?[Topics in Circuits for Communications]. *IEEE Communications Magazine* 45, 8 (2007).

[56] Per Stenstrom, Mats Brorsson, and Lars Sandberg. 1993. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *ISCA*.

[57] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-electronic Networks-on-Chip Modeling. In *NoCS*.

[58] Chen Sun, Mark T. Wade, Yunsup Lee, Jason S. Orcutt, Luca Alloatti, Michael S. Georgas, Andrew S. Waterman, Jeffrey M. Shainline, Rimas R. Avizienis, Sen Lin, Benjamin R. Moss, Rajesh Kumar, Fabio Pavanello, Amir H. Atabaki, Henry M. Cook, Albert J. Ou, Jonathan C. Leu, Yu-Hsin Chen, Krste Asanović, Rajeev J. Ram, Miloš A. Popović, and Vladimir M. Stojanović. 2015. Single-chip microprocessor that communicates directly using light. *Nature* 528, 7583 (2015).

[59] Guang Sun, Shih-Hung Weng, Chung-Kuan Cheng, Bill Lin, and Lieguang Zeng. 2012. 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*.

[60] Rafael Ubal, Perhaad Mistry, Dana Schaa, Huntington Ave, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT*.

[61] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: Exploiting Breakable Dependences for Parallelization. In *PLDI*.

[62] Dana Vantrease, Robert Schreiber, Matteo Monchiero, M. McLaren, N.P. Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, R.G. Beausoleil, and J.H. Ahn. 2008. Corona: System Implications of Emerging Nanophotonic Technology. In *ISCA*.

[63] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*.

[64] Benwei Xu, Yuan Zhou, and Yun Chiu. 2017. A 23-mW 24-GS/s 6-bit Voltage-Time Hybrid Time-Interleaved ADC in 28-nm CMOS. *IEEE Journal of Solid-State Circuits* 52, 4 (2017).

[65] Xinmin Yu, Joe Baylon, Paul Wettin, Deukhyoun Heo, Partha Pratim Pande, and Shahriar Mirabbasi. 2014. Architecture and Design of Multi-Channel Millimeter-Wave Wireless Network-on-Chip. *IEEE Design & Test* 31, 6 (2014).

[66] Xinmin Yu, Hooman Rashtian, and Shahriar Mirabbasi. 2015. 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 (2015).

[67] Xinmin Yu, Suman Prasad Sah, Hooman Rashtian, Shahriar Mirabbasi, Partha Pratim Pande, and Deukhyoun Heo. 2014. 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* 62, 10 (2014).

[68] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R Gao. 2007. Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures. In *ISCA*.