SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures

Christina Giannoula[†‡] Nandita Vijaykumar[*‡] Nikela Papadopoulou[†] Vasileios Karakostas[†] Ivan Fernandez[§‡]
Juan Gómez-Luna[‡] Lois Orosa[‡] Nectarios Koziris[†] Georgios Goumas[†] Onur Mutlu[‡]
[†]*National Technical University of Athens* [‡]*ETH Zürich* [*]*University of Toronto* [§]*University of Malaga*

Near-Data-Processing (NDP) architectures present a promising way to alleviate data movement costs and can provide significant performance and energy benefits to parallel applications. Typically, NDP architectures support several NDP units, each including multiple simple cores placed close to memory. To fully leverage the benefits of NDP and achieve high performance for parallel workloads, efficient synchronization among the NDP cores of a system is necessary. However, supporting synchronization in many NDP systems is challenging because they lack shared caches and hardware cache coherence support, which are commonly used for synchronization in multicore systems, and communication across different NDP units can be expensive.

This paper comprehensively examines the synchronization problem in NDP systems, and proposes SynCron, an end-to-end synchronization solution for NDP systems. SynCron adds low-cost hardware support near memory for synchronization acceleration, and avoids the need for hardware cache coherence support. SynCron has three components: 1) a specialized cache memory structure to avoid memory accesses for synchronization and minimize latency overheads, 2) a hierarchical message-passing communication protocol to minimize expensive communication across NDP units of the system, and 3) a hardware-only overflow management scheme to avoid performance degradation when hardware resources for synchronization tracking are exceeded.

We evaluate SynCron using a variety of parallel workloads, covering various contention scenarios. SynCron improves performance by 1.27× on average (up to 1.78×) under high-contention scenarios, and by 1.35× on average (up to 2.29×) under low-contention real applications, compared to state-of-the-art approaches. SynCron reduces system energy consumption by 2.08× on average (up to 4.25×).

1. Introduction

Recent advances in 3D-stacked memories [59, 72, 85, 92, 93, 145] have renewed interest in Near-Data Processing (NDP) [8, 9, 17, 110]. NDP involves performing computation close to where the application data resides. This alleviates the expensive data movement between processors and memory, yielding significant performance improvements and energy savings in parallel applications. Placing low-power cores or special-purpose accelerators (hereafter called NDP cores) close to the memory dies of high-bandwidth 3D-stacked memories is a commonly-proposed design for NDP systems [8, 9, 19–21, 23, 38, 42–46, 49, 66, 67, 82–84, 98, 105, 110–113, 117, 119, 131, 132, 143, 155, 158]. Typical NDP architectures support several NDP units connected to each other, with each unit comprising multiple NDP cores close to memory [8, 19, 66, 83, 143, 155, 158]. Therefore, NDP architectures provide high levels of parallelism, low memory access latency, and large aggregate memory bandwidth.

Recent research demonstrates the benefits of NDP for parallel applications, e.g., for genome analysis [23, 84], graph processing [8, 9, 20, 21, 112, 155, 158], databases [20, 38], security [54], pointer-chasing workloads [25, 60, 67, 99], and neural networks [19, 45, 82, 98]. In general, these applications exhibit high parallelism, low operational intensity, and relatively low cache locality [15, 16, 33, 50, 133], which make them suitable for NDP.

Prior works discuss the need for efficient synchronization primitives in NDP systems, such as locks [25, 99] and barriers [8, 43, 155, 158]. Synchronization primitives are widely used by multithreaded applications [39, 40, 48, 69, 70, 90, 136–138, 140], and must be carefully designed to fit the underlying hardware requirements to achieve high performance. Therefore, to fully leverage the benefits of NDP for parallel applications, an effective synchronization solution for NDP systems is necessary.

Approaches to support synchronization are typically of two types [63, 64]. First, synchronization primitives can be built through *shared memory*, most commonly using the atomic read-modify-write (*rmw*) operations provided by hardware. In CPU systems, atomic *rmw* operations are typically implemented upon the underlying hardware cache coherence protocols, but many NDP systems do *not* support hardware cache coherence (e.g., [8, 46, 143, 155, 158]). In GPUs and Massively Parallel Processing systems (MPPs), atomic *rmw* operations can be implemented in dedicated hardware atomic units, known as *remote atomics*. However, synchronization using remote atomics has been shown to be inefficient, since sending every update to a fixed location creates high global traffic and hotspots [41, 96, 108, 147, 153]. Second, synchronization can be implemented via a *message-passing* scheme, where cores exchange messages to reach an agreement. Some recent NDP works (e.g., [8, 43, 55, 158]) propose message-passing barrier primitives among NDP cores of the system. However, these synchronization schemes are still inefficient, as we demonstrate in Section 6, and also lack support for lock, semaphore and condition variable synchronization primitives.

Hardware synchronization techniques that do not rely on hardware coherence protocols and atomic *rmw* operations have been proposed for multicore systems [1–3, 94, 97, 116, 146, 157]. However, such synchronization schemes are tailored for the specific architecture of each system, and are not efficient or suitable for NDP systems (Section 7). For instance, CM5 [94] provides a barrier primitive via a dedicated physical network, which would incur high hardware cost to be supported in large-scale NDP systems. LCU [146] adds a control unit to *each* CPU core and a buffer to each memory controller, which would also incur high cost to implement in *area-constrained* NDP cores and controllers. SSB [157] includes a small buffer attached to each controller of the last level cache (LLC) and MiSAR [97] introduces an accelerator distributed at the LLC. Both schemes are built on the shared cache level in CPU systems, which most NDP systems do *not* have. Moreover, in NDP systems with *non-uniform* memory access times, most of these prior schemes would incur significant performance overheads under high-contention scenarios. This is because they are oblivious to the non-uniformity of NDP, and thus would cause excessive traffic across NDP units of the system upon contention (Section 6.7.1).

Overall, NDP architectures have several important characteristics that necessitate a new approach to support efficient synchronization. First, most NDP architectures [8, 19, 25, 38, 42–46, 49, 55, 67, 98, 110, 111, 113, 119, 155, 158] lack shared

caches that can enable low-cost communication and synchronization among NDP cores of the system. Second, hardware cache coherence protocols are typically not supported in NDP systems [8,19,25,38,42–45,49,55,67,82,98,111,119,155,158], due to high area and traffic overheads associated with such protocols [46, 143]. Third, NDP systems are non-uniform, distributed architectures, in which inter-unit communication is more expensive (both in performance and energy) than intra-unit communication [8, 20, 21, 38, 43, 83, 155, 158].

In this work, we present *SynCron*, an efficient synchronization mechanism for NDP architectures. *SynCron* is designed to achieve the goals of performance, cost, programming ease, and generality to cover a wide range of synchronization primitives through four key techniques. First, we offload synchronization among NDP cores to dedicated low-cost hardware units, called Synchronization Engines (SEs). This approach avoids the need for complex coherence protocols and expensive *rmw* operations, at low hardware cost. Second, we directly buffer the synchronization variables in a specialized cache memory structure to avoid costly memory accesses for synchronization. Third, *SynCron* coordinates synchronization with a hierarchical message-passing scheme: NDP cores only communicate with their local SE that is located in the same NDP unit. At the next level of communication, all local SEs of the system's NDP units communicate with each other to coordinate synchronization at a global level. Via its hierarchical communication protocol, *SynCron* significantly reduces synchronization traffic across NDP units under high-contention scenarios. Fourth, when applications with frequent synchronization oversubscribe the hardware synchronization resources, *SynCron* uses an efficient and programmer-transparent overflow management scheme that avoids costly fallback solutions and minimizes overheads.

We evaluate *SynCron* using a wide range of parallel workloads including pointer chasing, graph applications, and time series analysis. Over prior approaches (similar to [8, 43]), *SynCron* improves performance by 1.27× on average (up to 1.78×) under high-contention scenarios, and by 1.35× on average (up to 2.29×) under low-contention scenarios. In real applications with fine-grained synchronization, *SynCron* comes within 9.5% of the performance and 6.2% of the energy of an ideal zero-overhead synchronization mechanism. Our proposed hardware unit incurs very modest area and power overheads (Section 6.8) when integrated into the compute die of an NDP unit.

This paper makes the following contributions:

- We investigate the challenges of providing efficient synchronization in Near-Data-Processing architectures, and propose an end-to-end mechanism, *SynCron*, for such systems.
- We design low-cost synchronization units that coordinate synchronization across NDP cores, and directly buffer synchronization variables to avoid costly memory accesses to them. We propose an efficient message-passing synchronization approach that organizes the process hierarchically, and provide a hardware-only programmer-transparent overflow management scheme to alleviate performance overheads when hardware synchronization resources are exceeded.
- We evaluate *SynCron* using a wide range of parallel workloads and demonstrate that it significantly outperforms prior approaches both in performance and energy consumption. *SynCron* also has low hardware area and power overheads.

2. Background and Motivation

2.1. Baseline Architecture

Numerous works [8, 9, 19–21, 25, 38, 43, 45, 54, 55, 67, 73, 82, 99, 112, 128, 143, 155, 158] show the potential benefit of NDP for parallel, irregular applications. These proposals focus on the design of the compute logic that is placed close to or within memory, and in many cases provide special-purpose near-data accelerators for specific applications. Figure 1 shows the baseline organization of the NDP architecture we assume in this work, which includes several NDP units connected with each other via serial interconnection links to share the same physical address space. Each NDP unit includes the memory arrays and a compute die with multiple low-power programmable cores or fixed-function accelerators, which we henceforth refer to as NDP cores. NDP cores execute the offloaded NDP kernel and access the various memory locations across NDP units with non-uniform access times [8, 20, 21, 38, 143, 155, 158]. We assume that there is no OS running in the NDP system. In our evaluation, we use programmable in-order NDP cores, each including small private L1 I/D caches. However, *SynCron* can be used with any programmable, fixed-function or reconfigurable NDP accelerator. We assume software-assisted cache-coherence (provided by the operating system or the programmer), similar to [43, 143]: data can be either thread-private, shared read-only, or shared read-write. Thread-private and shared read-only data can be cached by NDP cores, while shared read-write data is uncacheable.

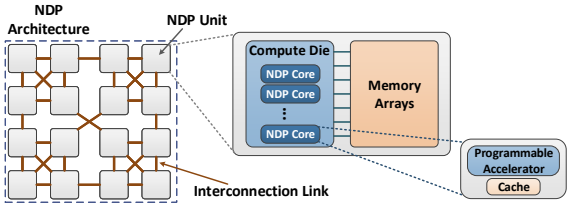


Figure 1: High-level organization of an NDP architecture.

We focus on three characteristics of NDP architectures that are of particular importance in the synchronization context. First, NDP architectures typically do not have a shared level of cache memory [8, 19, 25, 38, 42–46, 49, 55, 67, 98, 110, 111, 113, 119, 155, 158], since the NDP-suited workloads usually do not benefit from deep cache hierarchies due to their poor locality [33, 43, 133, 143]. Second, NDP architectures do not typically support conventional hardware cache coherence protocols [8,19,25,38,42–45,49,55,67,82,98,111,119,155,158], because they would add area and traffic overheads [46, 143], and would incur high complexity and latency [4], limiting the benefits of NDP. Third, communication across NDP units is expensive, because NDP systems are non-uniform distributed architectures. The energy and performance costs of inter-unit communication are typically orders of magnitude greater than the costs of intra-unit communication [8,20,21,38,43,83,155,158], and thus inter-unit communication may slow down the execution of NDP cores [155].

2.2. The Solution Space for Synchronization

Approaches to support synchronization are typically either via shared memory or message-passing schemes.

2.2.1. Synchronization via Shared Memory. In this case, cores coordinate via a consistent view of shared memory locations, using atomic read/write operations or atomic read-modify-write (*rmw*) operations. If *rmw* operations are *not* supported by hardware, Lamport's bakery algorithm [87] can provide synchronization to N participating cores, assuming sequential consistency [86]. However, this scheme scales poorly, as a core accesses $O(N)$ memory locations at *each* synchronization retry. In contrast, commodity systems (CPUs, GPUs, MPPs) typically support *rmw* operations in hardware.

GPUs and MPPs support *rmw* operations in specialized hardware units (known as *remote atomics*), located in each bank of the shared cache [58, 148], or the memory controllers [81,88]. Remote atomics are also supported by an NDP

work [43] at the vault controllers of Hybrid Memory Cube (HMC) [59,145]. Implementing synchronization primitives using remote atomics requires a spin-wait scheme, i.e., executing consecutive *rmw* retries. However, performing and sending every *rmw* operation to a shared, fixed location can cause high global traffic and create hotspots [41, 96, 108, 147, 153]. In NDP systems, consecutive *rmw* operations to a remote NDP unit would incur high traffic *across* NDP units, with high performance and energy overheads.

Commodity CPU architectures support *rmw* operations either by locking the bus (or equivalent link), or by relying on the hardware cache coherence protocol [68, 135], which many NDP architectures do not support. Therefore, coherence-based synchronization [13, 24, 27, 35, 36, 57, 100, 101, 103, 122, 126, 156] cannot be directly implemented in NDP architectures. Moreover, based on prior works on synchronization [22, 30, 76, 102, 107, 140], coherence-based synchronization would exhibit low scalability on NDP systems for two reasons. First, it performs poorly with a *large* number of cores, due to low scalability of conventional hardware coherence protocols [61, 79, 80, 135]. Most NDP systems include several NDP units [8, 83, 155, 158], each typically supporting hundreds of small, area-constrained cores [8, 19, 155, 158]. Second, the non-uniformity in memory accesses significantly affects the scalability of coherence-based synchronization [22,30,107,156]. Prior work on coherence-based synchronization [30] observes that the latency of a lock acquisition that needs to transfer the lock *across* NUMA sockets can be up to 12.5× higher than that *within* a socket. We expect such effects to be aggravated in NDP systems, since they are by nature *non-uniform* and *distributed* [8,20,21,38,43,83,155,158] with very low memory access latency within an NDP unit.

We validate these observations on both a real CPU and our simulated NDP system. On an Intel Xeon Gold server, we evaluate the operation throughput achieved by two coherence-based lock algorithms (Table 1), i.e., TTAS [122] and Hierarchical Ticket Lock (HTL) [103], using a microbenchmark taken from the *libslock* library [30]. When increasing the number of threads from 1 to 14 within a single socket, throughput drops by 3.91× and 2.77× for TTAS and HTL, respectively. Moreover, when pinning two threads on different NUMA sockets, throughput drops by up to 2.29× over when pinning them on the same socket, due to non-uniform memory access times of lock variables.

Million Operations per Second	1 thread single-socket	14 threads single-socket	2 threads same-socket	2 threads different-socket
TTAS lock [122]	8.92	2.28	9.91	4.32
Hierarchical Ticket lock [103]	8.06	2.91	9.01	6.79

Table 1: Throughput of two coherence-based lock algorithms on an Intel Xeon Gold server using the libslock library [30].

In our simulated NDP system, we evaluate the performance achieved by a stack data structure protected with a coarse-grained lock. Figure 2 shows the slowdown of the stack when using a coherence-based lock [63] (*mesi-lock*), implemented upon a MESI directory coherence protocol, over using an ideal lock with zero cost for synchronization (*ideal-lock*). First, we observe that the high contention for the cache line containing the *mesi-lock* and the resulting coherence traffic inside the network significantly limit scalability of the stack as the number of cores increases. With 60 NDP cores within a single NDP unit (Figure 2a), the stack with *mesi-lock* incurs 2.03× slowdown over *ideal-lock*. Second, we notice that the non-uniform memory accesses to the cache line containing the *mesi-lock* also impact the scalability of the stack. When increasing the number of NDP units while keeping total core count constant at 60 (Figure 2b), the slowdown of the stack with *mesi-lock* increases to 2.66× (using 4 NDP units) over *ideal-lock*. In *non-uniform* NDP systems, the scalability of coherence-based synchronization is severely limited by the long transfer latency and low bandwidth of the interconnect used between the NDP units.

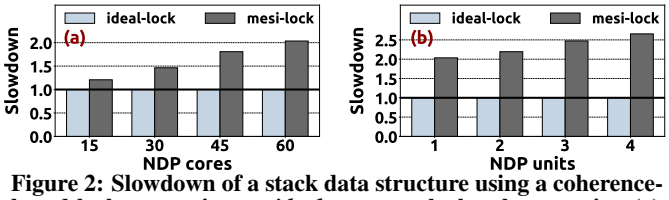


Figure 2: Slowdown of a stack data structure using a coherence-based lock over using an *ideal* zero-cost lock, when varying (a) the NDP cores within a single NDP unit and (b) the number of NDP units while keeping core count constant at 60.

2.2.2. Message-passing Synchronization. In this approach, cores coordinate with each other by exchanging messages (either in software or hardware) in order to reach an agreement. For instance, a recent NDP work [8] implements a barrier primitive via hardware message-passing communication among NDP cores, i.e., one core of the system works as a *master* core to collect the synchronization status of the rest. To improve system performance in *non-uniform* HMC-based NDP systems, Gao et al. [43] propose a *tree-style* barrier primitive, where cores exchange messages to first synchronize within a vault, then across the vaults of an HMC cube, and finally across HMC cubes. In general, optimized message-passing synchronization schemes proposed in the literature [2,43,53,62,64,141] aim to minimize (i) the number of messages sent among cores, and (ii) expensive network traffic. To avoid the major issues of synchronization via shared memory described above, we design our approach building on the message-passing synchronization concept.

3. *SynCron*: Overview

SynCron is an end-to-end solution for synchronization in NDP architectures that improves performance, has low cost, eases programmability, and supports multiple synchronization primitives. *SynCron* relies on the following key techniques:

1. Hardware support for synchronization acceleration: We design low-cost hardware units, called Synchronization Engines (SEs), to coordinate the synchronization among NDP cores of the system. SEs eliminate the need for complex cache coherence protocols and expensive *rmw* operations, and incur modest hardware cost.

2. Direct buffering of synchronization variables: We add a specialized cache structure, the Synchronization Table (ST), inside an SE to keep synchronization information. Such direct buffering avoids costly memory accesses for synchronization, and enables high performance under low-contention scenarios.

3. Hierarchical message-passing communication: We organize the communication hierarchically, with each NDP unit including an SE. NDP cores communicate with their local SE that is located in the same NDP unit. SEs communicate with each other to coordinate synchronization at a global level. Hierarchical communication minimizes expensive communication *across* NDP units, and achieves high performance under high-contention scenarios.

4. Integrated hardware-only overflow management: We incorporate a hardware-only overflow management scheme to efficiently handle scenarios when ST is fully occupied. This programmer-transparent technique effectively limits performance degradation under overflow scenarios.

3.1. Overview of *SynCron*

Figure 3 provides an overview of our approach. *SynCron* exposes a simple programming interface such that programmers can easily use a variety of synchronization primitives in their

multithreaded applications when writing them for NDP systems. The interface is implemented using two new instructions that are used by NDP cores to communicate synchronization requests to SEs. These are general enough to cover all semantics for the most widely-used synchronization primitives.

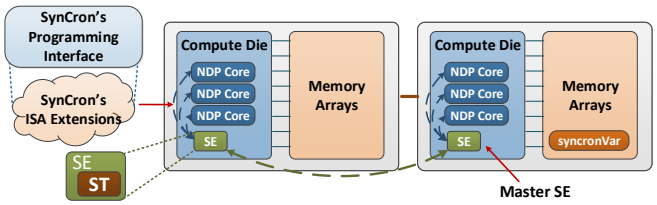


Figure 3: High-level overview of *SynCron*.

We add one SE in the compute die of each NDP unit. For a particular synchronization variable allocated in an NDP unit, the SE that is physically located in the same NDP unit is considered the *Master SE*. In other words, the *Master SE* is defined by the address of the synchronization variable. It is responsible for the global coordination of synchronization on that variable, i.e., among all SEs of the system. All other SEs are responsible only for the local coordination of synchronization among the cores in the same NDP unit with them.

NDP cores act as clients that send requests to SEs via hardware message-passing. SEs act as servers that process synchronization requests. In the proposed hierarchical communication, NDP cores send requests to their local SEs, while SEs of different NDP units communicate with the *Master SE* of the specific variable, to coordinate the process at a global level, i.e., among all NDP units.

When an SE receives a request from an NDP core for a synchronization variable, it directly buffers the variable in its ST, keeping all the information needed for synchronization in the ST. If the ST is full, we use the main memory as a fallback solution. To hierarchically coordinate synchronization via main memory in ST overflow cases, we design (i) a generic structure, called *syncronVar*, to keep track of required synchronization information, and (ii) specialized *overflow* messages to be sent among SEs. The hierarchical communication among SEs is implemented via corresponding support in message encoding, the ST, and *syncronVar* structure.

3.2. *SynCron*'s Operation

SynCron supports locks, barriers, semaphores, and condition variables. Here, we present *SynCron*'s operation for locks. *SynCron* has similar behavior for the other three primitives.

Lock Synchronization Primitive: Figure 4 shows a system composed of two NDP units with two NDP cores each. In this example, all cores request and compete for the same lock. First, all NDP cores send *local* lock acquire messages to their SEs ❶. After receiving these messages, each SE keeps track of its requesting cores by reserving one new entry in its ST, i.e., directly buffering the lock variable in ST. Each ST entry includes a local waiting list (i.e., a hardware bit queue with one bit for each local NDP core), and a global waiting list (i.e., a bit queue with one bit for each SE of the system). To keep track of the requesting cores, each SE sets the bits corresponding to the requesting cores in the local waiting list of the ST entry. When the local SE receives a request for a synchronization variable *for the first time*, it sends a *global* lock acquire message to the *Master SE* ❷, which in turn sets the corresponding bit in the global waiting list in its ST. This way, the *Master SE* keeps track of all requests to a particular variable coming from an SE, and can arbitrate between different SEs. The local SE can then serve successive local requests to the same variable until there are no other local requests. By using the proposed hierarchical communication protocol, the cores send local messages to their local SE, and the SE needs to send *only one aggregated* message, on behalf of all its local waiting cores, to the *Master SE*. As a result, we reduce the need for communication through the narrow, expensive links that connect different NDP units.

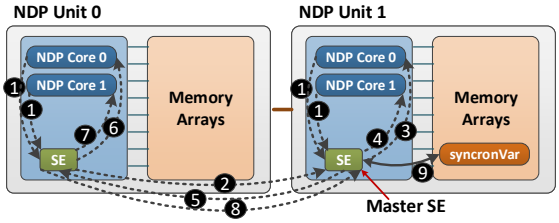


Figure 4: An example execution scenario for a lock requested by *all* NDP cores.

The *Master SE* first prioritizes the local waiting list, granting the lock to its own local NDP cores in sequence (e.g., to NDP Core 0 first ❸, and to NDP Core 1 next ❹ in Figure 4). At the end of the critical section, each local lock owner sends a lock release message to its SE in order to release the lock. When there are no other local requests, the *Master SE* transfers the control of the lock to the SE of another NDP unit based on its global waiting list ❺. Then, the local SE grants the lock to its local NDP cores in sequence (e.g., ❻, ❼). After all local cores release the lock, the SE sends an *aggregated* global lock release message to the *Master SE* ❽ and releases its ST entry. When the message arrives at the *Master SE*, if there are no other pending requests to the same variable, the *Master SE* releases its ST entry. In this example, SEs directly buffer the lock variable in their STs. If an ST is *full*, the *Master SE* globally coordinates synchronization by keeping track of all required information in main memory ❾, via our proposed overflow management scheme (Section 4.3).

4. *SynCron*: Detailed Design

SynCron leverages the key observation that all synchronization primitives fundamentally communicate the same information, i.e., a waiting list of cores that participate in the synchronization process, and a condition to be met to notify one or more cores. Based on this observation, we design *SynCron* to cover the four most widely used synchronization primitives. Without loss of generality, we assume that each NDP core represents a hardware thread context with a unique ID. To support multiple hardware thread contexts per NDP core, the corresponding hardware structures of *SynCron* need to be augmented to include 1-bit per hardware thread context.

4.1. Programming Interface and ISA Extensions

SynCron provides lock, barrier, semaphore and condition variable synchronization primitives, supporting two types of barriers: within cores of the *same* NDP unit and within cores across different NDP units of the system. *SynCron*'s programming interface (Table 2) implements the synchronization semantics with two new ISA instructions, which are *rich* and *general* enough to express all supported primitives. NDP cores use these instructions to assemble messages for synchronization requests, which are issued through the network to SEs.

***SynCron* Programming Interface**
syncronVar *create_syncvar ();
void destroy_syncvar (syncronVar *svar);
void lock_acquire (syncronVar *lock);
void lock_release (syncronVar *lock);
void barrier_wait_within_unit (syncronVar *bar, int initialCores);
void barrier_wait_across_units (syncronVar *bar, int initialCores);
void sem_wait (syncronVar *sem, int initialResources);
void sem_post (syncronVar *sem);
void cond_wait (syncronVar *cond, syncronVar *lock);
void cond_signal (syncronVar *cond);
void cond_broadcast (syncronVar *cond);

Table 2: *SynCron*'s Programming Interface (i.e., API).

req_sync addr, opcode, info: This instruction creates a message and commits when a response message is received back. The *addr* register has the address of a synchronization variable, the *opcode* register has the message opcode of a particular semantic of a synchronization primitive (Table 3), and the *info* register has specific information needed for the primitive (*MessageInfo* in message encoding of Fig. 5).

req_async addr, opcode: This instruction creates a message and after the message is issued to the network, the instruction commits. The registers *addr*, *opcode* have the same semantics as in *req_sync* instruction.

4.1.1. Memory Consistency. We design *SynCron* assuming a relaxed consistency memory model. The proposed ISA extensions act as memory fences. First, *req_sync*, commits once a message (ACK) is received (from the local SE to the core), which ensures that all following instructions will be issued after *req_sync* has been completed. Its semantics is similar to those of the SYNC and ACQUIRE operations of Weak Ordering (WO) [28] and Release Consistency (RC) [28] models, respectively. Second, *req_async*, does not require a return message (ACK). It is issued once all previous instructions are completed. Its semantics is similar to that of the RELEASE operation of RC [28]. In the case of WO, *req_sync* is sufficient. In the case of RC, the *req_sync* instruction is used for acquire-type semantics, i.e., lock_acquire, barrier_wait, semaphore_wait and condition_variable_wait, while the *req_async* instruction is used for release-type semantics, i.e., lock_release, semaphore_post, condition_variable_signal, and condition_variable_broadcast.

4.1.2. Message Encoding. Figure 5 describes the encoding of the message used for communication between NDP cores and the SE. Each message includes: (i) the 64-bit address of the synchronization variable, (ii) the message opcode that implements the semantics of the different synchronization primitives (6 bits cover all message opcodes), (iii) the unique ID number of the NDP core (6 bits are sufficient for our simulated NDP system in Section 5), and (iv) a 64-bit field (*MessageInfo*) that communicates specific information needed for each different synchronization primitive, i.e., the number of the cores that participate in a barrier, the initial value of a semaphore, the address of the lock associated with a condition variable.

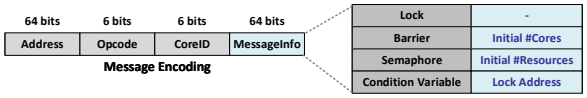


Figure 5: Message encoding of *SynCron*.

Hierarchical Message Opcodes. *SynCron* enables a hierarchical scheme, where the SEs of NDP units communicate with each other to coordinate synchronization at a global level. Therefore, we support two types of messages (Table 3): (i) *local*, which are used by NDP cores to communicate with their local SE, and (ii) *global*, which are used by SEs to communicate with the *Master SE*, and vice versa. Since we support two types of barriers (Table 2), we design two message opcodes for a *local* barrier_wait message sent by an NDP core to its local SE: (i) *barrier_wait_local_within_unit* is used when cores of a single NDP unit participate in the barrier, and (ii) *barrier_wait_local_across_units* is used when cores from different NDP units participate in the barrier. In the latter case, if a *smaller* number of cores than the total *available* cores of the NDP system participate in the barrier, *SynCron* supports one-level communication: local SEs re-direct all messages (received from their local NDP cores) to the *Master SE*, which globally coordinates the barrier among *all* participating cores. This design choice is a trade-off between performance (*more remote messages*) and hardware/ISA complexity, since the number of participating cores of *each* NDP unit would need to be communicated to the hardware through additional registers in ISA, and message opcodes (*higher complexity*).

Primitives	*SynCron* Message Opcodes
Locks	lock_acquire_global, lock_acquire_local, lock_release_global lock_release_local, lock_grant_global, lock_grant_local lock_acquire_overflow, lock_release_overflow, lock_grant_overflow
Barriers	barrier_wait_global, barrier_wait_local_within_unit barrier_wait_local_across_units, barrier_depart_global, barrier_depart_local barrier_wait_overflow, barrier_departure_overflow
Semaphores	sem_wait_global, sem_wait_local, sem_grant_global sem_grant_local, sem_post_global, sem_post_local sem_wait_overflow, sem_grant_overflow, sem_post_overflow
Condition Variables	cond_wait_global, cond_wait_local, cond_signal_global cond_signal_local, cond_broad_global, cond_broad_local cond_grant_global, cond_grant_local, cond_wait_overflow cond_signal_overflow, cond_broad_overflow, cond_grant_overflow
Other	decrease_indexing_counter

Table 3: Message opcodes of *SynCron*.

4.2. Synchronization Engine (SE)

Each SE module (Figure 6) is integrated into the compute die of each NDP unit. An SE consists of *three* components:

4.2.1. Synchronization Processing Unit (SPU). The SPU is the logic that handles the messages, updates the ST, and issues requests to memory as needed. The SPU includes the control unit, a buffer, and a few registers. The buffer is a small SRAM queue for temporarily storing messages that arrive at the SE. The control unit implements custom logic with simple logical bitwise operators (and, or, xor, zero) and multiplexers.

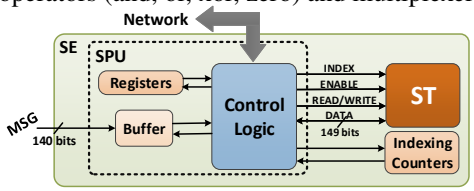


Figure 6: The Synchronization Engine (SE).

4.2.2. Synchronization Table (ST). ST keeps track of all the information needed to coordinate synchronization. Each ST has 64 entries. Figure 7 shows an ST entry, which includes: (i) the 64-bit address of a synchronization variable, (ii) the global waiting list used by the *Master SE* for global synchronization among SEs, i.e., a hardware bit queue including one bit for each SE of the system, (iii) the local waiting list used by all SEs for synchronization among the NDP cores of an NDP unit, i.e., a hardware bit queue including one bit for each NDP core within the unit, (iv) the state of the ST entry, which can be either *free* or *occupied*, and (v) a 64-bit field (*TableInfo*) to track specific information needed for each synchronization primitive. For the lock primitive, the *TableInfo* field is used to indicate the lock owner that is either an SE of an NDP unit (*Global ID* represented by the most significant bits) or a *local* NDP core (*Local ID* represented by the least significant bits). We assume that all NDP cores of an NDP unit have a unique *local ID* within the NDP unit, while all SEs of the system have a unique *global ID* within the system. The number of bits in the global and local waiting lists of Figure 7 is specific for the configuration of our evaluated system (Section 5), which includes 16 NDP cores per NDP unit and 4 SEs (one per NDP unit), and has to be extended accordingly, if the system supports more NDP cores or SEs.

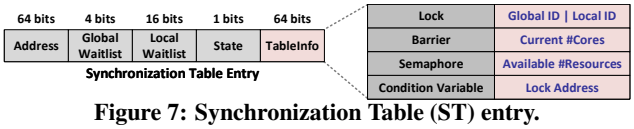


Figure 7: Synchronization Table (ST) entry.

4.2.3. Indexing Counters. If an ST is full, i.e., all its entries are in *occupied* state, *SynCron* cannot keep track of information for a new synchronization variable in ST. We use the main memory as a fallback solution for such ST overflow

(Section 4.3). The SE keeps track of *which* synchronization variables are currently serviced via main memory: similar to MiSAR [97], we include a small set of counters (*indexing counters*), 256 in current implementation, indexed by the least significant bits of the address of a synchronization variable, as extracted from the message that arrives at an SE. When an SE receives a message with acquire-type semantics for a synchronization variable and there is no corresponding entry in the *fully-occupied* ST, the indexing counter for that synchronization variable increases. When an SE receives a message with release-type semantics for a synchronization variable that is currently serviced using main memory, the corresponding indexing counter decreases. A synchronization variable is currently serviced via main memory, when the corresponding indexing counter is larger than zero. Note that different variables may alias to the same indexing counter. This aliasing does not affect correctness, but it does affect performance, since a variable may unnecessarily be serviced via main memory, while the ST is *not* full.

4.2.4. Control Flow in SE. Figure 8 describes the control flow in SE. When an SE receives a message, it decodes the message ❶ and accesses the ST ❷a. If there is an ST entry for the specific variable (depending on its address), the SE processes the waiting lists ❸, updates the ST ❹a, and encodes return message(s) ❺, if needed. If there is *not* an ST entry for the specific variable, the SE checks the value of the corresponding indexing counter ❷b: (i) if the indexing counter is zero *and* the ST is not full, the SE reserves a *new* ST entry and continues with step ❸, otherwise (ii) if the indexing counter is larger than zero *or* the ST is full, there is an overflow. In that case, if the SE is the *Master SE* for the specific variable, it reads the synchronization variable from *local* memory arrays ❷c, processes the waiting lists ❸, updates the variable in main memory ❹b, and encodes return message(s) ❺, if needed. If the SE is *not* the *Master SE* for the specific variable, it encodes an *overflow* message to the *Master SE* ❷d to handle overflow.

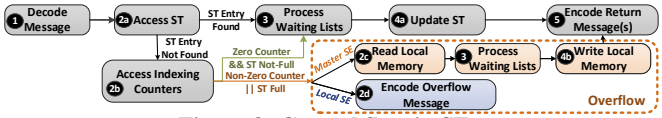


Figure 8: Control flow in SE.

4.3. Overflow Management

SynCron integrates a hardware-only overflow management scheme that provides very modest performance degradation (Section 6.7.3) and is programmer-transparent. To handle ST overflow cases, we need to address two issues: (i) where to keep track of required information to coordinate synchronization, and (ii) how to coordinate ST overflow cases between SEs. For the former issue, we design a generic structure allocated in main memory. For the latter issue, we propose a hierarchical *overflow* communication protocol between SEs.

4.3.1. *SynCron*'s Synchronization Variable. We design a generic structure (Figure 9), called *syncronVar*, which is used to coordinate synchronization for all supported primitives in ST overflow cases. *syncronVar* is defined in the driver of the NDP system, which handles the allocation of the synchronization variables: programmers use *create_syncvar()* (Table 2) to create a *new* synchronization variable, the driver allocates the bytes needed for *syncronVar* in main memory, and returns an opaque pointer that points to the address of the variable. Programmers should not de-reference the opaque pointer and its content can *only* be accessed via *SynCron*'s API (Table 2).

syncronVar structure includes one waiting list for each SE of the system, which has one bit for each NDP core within the

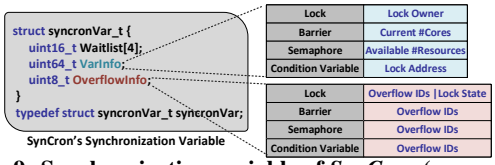


Figure 9: Synchronization variable of *SynCron* (*syncronVar*).

NDP unit, and two additional fields (*VarInfo, OverflowInfo*) needed to hierarchically handle ST overflows for all primitives.

4.3.2. Communication Protocol between SEs. To ensure correctness, *only* the *Master SE* updates the *syncronVar* variable: in ST overflow, the SPU of the *Master SE* issues read or write requests to its local memory to *globally* coordinate synchronization via the *syncronVar* variable. In our proposed hierarchical design, there are two overflow scenarios: (i) the ST of the *Master SE* overflows, and (ii) the ST of a local SE overflows or STs of multiple local SEs overflow.

The ST of the *Master SE* overflows. The other SEs of the system have *not* overflowed for a specific synchronization variable. Thus, they can still directly buffer this variable in their local STs, and serve their local cores themselves, implementing a hierarchical (two-level) communication with *Master SE*. The *Master SE* receives *global* messages from SEs, and serves a local SE of an NDP unit using *all* bits in the waiting list of the *syncronVar* variable associated with that local SE. Specifically, when it receives a *global* acquire-type message from a local SE, it sets *all* bits in the corresponding waiting list of the *syncronVar* variable. When it receives a *global* release-type message from a local SE, it resets *all* bits in the corresponding waiting list of the *syncronVar* variable.

The ST of a local SE overflows. In this scenario, there are local SEs that have overflowed for a specific variable, and local SEs that have *not* overflowed. Without loss of generality, we assume that only one SE of the system has overflowed.

The local SEs that have *not* overflowed serve their local cores themselves via their STs, implementing a hierarchical (two-level) communication with *Master SE*. When the *Master SE* receives a *global* message from a local SE (that has *not* overflowed), it (i) sets (or resets) *all bits* in the waiting list of the *syncronVar* variable associated with that SE, and (ii) responds with a *global* message to the local SE, if needed.

The overflowed SE needs to notify the *Master SE* to handle *local* synchronization requests of NDP cores located at *another* NDP unit via main memory. We design *overflow* message opcodes (Table 3) to be sent from the local overflowed SE to the *Master SE* and back. The overflowed SE re-directs all messages (sent from its local NDP cores) for a specific variable to the *Master SE* using the *overflow* message opcodes, and both the overflowed SE and the *Master SE* increase their corresponding indexing counters to indicate that this variable is currently serviced via memory. When the *Master SE* receives an *overflow* message, it (i) sets (or resets) in the waiting list (associated with the overflowed SE) of the *syncronVar* variable, the bit that corresponds to the *local ID* of the NDP core within the NDP unit, (ii) sets (or resets) in the *OverflowInfo* field of the *syncronVar* variable the bit that corresponds to the *global ID* of the overflowed SE to keep track of *which* SE (or SEs) of the system has overflowed, and (iii) responds with an *overflow* message to that SE, if needed. The *local ID* of the NDP core, and the *global ID* of the overflowed SE are encoded in the *CoreID* field of the message (Figure 5). When all bits in the waiting lists of the *syncronVar* variable become zero (upon receiving a release-type message), the *Master SE* decrements the corresponding indexing counter. Then, it sends a *decrease_index_counter* message (Table 3) to the overflowed SE (based on the set bit that is tracked in the *OverflowInfo* field), which decrements its corresponding indexing counter.

4.4. *SynCron* Enhancements

4.4.1. *RMW* Operations. It is straightforward to extend *SynCron* to support simple atomic *rmw* operations inside the SE (by adding a lightweight ALU). The *Master SE* could be responsible for executing atomic *rmw* operations on a variable depending on its address. We leave that for future work.

4.4.2. Lock Fairness. When local cores of an NDP unit repeatedly request a lock from their local SE, the SE repeatedly grants the lock within its unit, potentially causing unfairness and delay to other NDP units. To prevent this, an extra field of a local grant counter could be added to the ST entry. The counter increases every time the SE grants the lock to a local core. If the counter exceeds a predefined threshold, then when the SE receives a lock release, it transfers the lock to another SE (assuming other SEs request the lock). The host OS or the user could dynamically set this threshold via a dedicated register. We leave the exploration of such fairness mechanisms to future work.

4.5. Comparison with Prior Work

SynCron's design shares some of its design concepts with SSB [157], LCU [146], and MiSAR [97]. However, *SynCron* is more general, supporting the four most widely-used synchronization primitives, and easy-to-use thanks to its high-level programming interface.

Table 4 qualitatively compares *SynCron* with these schemes. SSB and LCU support only lock semantics, thus they introduce two *ISA extensions* for a simple lock. MiSAR introduces seven ISA extensions to support three primitives and handle overflow scenarios. *SynCron* includes two ISA extensions for four *supported primitives*. A *spin-wait approach* performs consecutive synchronization retries, typically incurring high energy consumption. A *direct notification* scheme sends a direct message to only one waiting core when the synchronization variable becomes available, minimizing the traffic involved upon a release operation. SSB, LCU and MiSAR are tailored for *uniform* memory systems. In contrast, *SynCron* is the *only* hardware synchronization mechanism that targets NDP systems as well as *non-uniform* memory systems.

SSB and LCU handle *overflow* in hardware synchronization resources using a pre-allocated table in main memory, and if it overflows, they switch to software exception handlers (handled by the programmer), which typically incur large overheads (due to OS intervention) when overflows happen at a non-negligible frequency. To avoid falling back to main memory, which has high latency, and using expensive software exception handlers, MiSAR requires the programmer to handle overflow scenarios using alternative software synchronization libraries (e.g., pthread library provided by the OS). This approach can provide performance benefits in CPU systems, since alternative synchronization solutions can exploit low-cost accesses to caches and hardware cache coherence. However, in NDP systems alternative solutions would by default use main memory due to the absence of shared caches and hardware cache coherence support. Moreover, when overflow occurs, MiSAR's accelerator sends abort messages to all participating CPU cores notifying them to use the alternative solution, and when the cores finish synchronizing via the alternative solution, they notify MiSAR's accelerator to switch back to hardware synchronization. This scheme introduces additional hardware/ISA complexity, and communication between the cores and the accelerator, thus incurring high network traffic and communication costs, as we show in Section 6.7.3. In contrast, *SynCron* directly falls back to memory via a fully-integrated hardware-only overflow scheme, which provides graceful performance degradation (Section 6.7.3), and is completely transparent to the programmer: programmers *only* use *SynCron*'s high-level API, similarly to how software libraries are in charge of synchronization.

	SSB [157]	LCU [146]	MiSAR [97]	**SynCron**
Supported Primitives	1	1	3	**4**
ISA Extensions	2	2	7	**2**
Spin-Wait Approach	yes	yes	no	**no**
Direct Notification	no	yes	yes	**yes**
Target System	uniform	uniform	uniform	**non-uniform**
Overflow Management	partially integrated	partially integrated	handled by programmer	**fully integrated**

Table 4: Comparison of *SynCron* with prior mechanisms.

4.6. Use of *SynCron* in Conventional Systems

The baseline NDP architecture [8, 43, 143, 155, 158] we assume in this work shares key design principles with conventional NUMA systems. However, unlike NDP systems, NUMA CPU systems (i) have a shared level of cache (within a NUMA socket and/or across NUMA sockets), (ii) run multiple multi-threaded applications, i.e., a high number of software threads executed in hardware thread contexts, and (iii) the OS migrates software threads between hardware thread contexts to improve system performance. Therefore, although *SynCron* could be implemented in such commodity systems, our proposed hardware design would need extensions. First, *SynCron* could exploit the low-cost accesses to *shared* caches in conventional CPUs, e.g., including an additional level in *SynCron*'s hierarchical design to use the shared cache for efficient synchronization within a NUMA socket, and/or handling overflow scenarios by falling back to the low-latency cache instead of main memory. Second, *SynCron* needs to support use cases (ii) and (iii) listed above in such systems, i.e., including larger STs and waiting lists to satisfy the needs of multiple multithreaded applications, handling the OS thread migration scenarios across hardware thread contexts, and handling multiple synchronization requests sent from different software threads with the same hardware ID to SEs, when different software threads are executed on the same hardware thread context. We leave the optimization of *SynCron*'s design for conventional systems to future work.

5. Methodology

Simulation Methodology. We use an in-house simulator that integrates ZSim [125] and Ramulator [85]. We model 4 NDP units (Table 5), each with 16 in-order cores. The cores issue a memory operation after the previous one has completed, i.e., there are no overlapping operations issued by the same core. Any write operation is completed (and the latency is accounted for in our simulations) before executing the next instruction. To ensure memory consistency, compiler support [123] guarantees that there is no reordering around the *sync* instructions and a read is inserted after a write inside a critical section.

NDP Cores	16 in-order cores @2.5 GHz per NDP unit
L1 Data + Inst. Cache	private, 16KB, 2-way, 4-cycle; 64 B line; 23/47 pJ per hit/miss [109]
NDP Unit Local Network	buffered crossbar network with packet flow control; 1-cycle arbiter; 1-cycle per hop [6]; 0.4 pJ/bit per hop [149]; M/D/1 model [18] for queueing latency;
DRAM HBM	4 stacks; 4GB HBM 1.0 [92, 93]; 500MHz with 8 channels; nRCDR/nRCDW/nRAS/nWR 7/6/17/8 ns [47, 85]; 7 pJ/bit [151]
DRAM HMC	4 stacks; 4GB HMC 2.1; 1250MHz; 32 vaults per stack; nRCD/nRAS/nWR 17/34/19 ns [47, 85]
DRAM DDR4	4 DIMMs; 4GB each DIMM DDR4 2400MHz; nRCD/nRAS/nWR 16/39/18 ns [47, 85]
Interconnection Links Across NDP Units	12.8GB/s per direction; 40 ns per cache line; 20-cycle; 4 pJ/bit
Synchronization Engine	SPU @1GHz clock frequency [129]; 8× 64-bit registers; buffer: 280B; ST: 1192B, 64 entries, 1-cycle [109]; indexing counters: 2304B, 256 entries (8 LSB of the address), 2-cycle [109]

Table 5: Configuration of our simulated system.

We evaluate three NDP configurations for different memory technologies, namely 2D, 2.5D, 3D NDP. The 2D NDP configuration uses a DDR4 memory model and resembles recent 2D NDP systems [34, 50, 89, 144]. In the 2.5D NDP

configuration, each compute die of NDP units (16 NDP cores) is connected to an HBM stack via an interposer, similar to current GPUs [106, 115] and FPGAs [131, 150]. For the 3D NDP configuration, we use the HMC memory model, where the compute die of the NDP unit is located in the logic layer of the memory stack, as in prior works [8, 19, 155, 158]. Due to space limitations, we present detailed evaluation results for the 2.5D NDP configuration, and provide a sensitivity study for the different NDP configurations in Section 6.5.

We model a crossbar network within each NDP unit, simulating queuing latency using the M/D/1 model [18]. We count in ZSim-Ramulator all events for caches, i.e., number of hits/misses, network, i.e., number of bits transferred inside/across NDP units, and memory, i.e., number of total memory accesses, and use CACTI [109] and parameters reported in prior works [143, 149, 151] to calculate energy. To estimate the latency in SE, we use CACTI for ST and indexing counters, and Aladdin [129] for the SPU with 1GHz at 40nm. Each message is served in 12 cycles, corresponding to the message (barrier_depart_global) that takes the longest time.

Workloads. We evaluate workloads with both (i) coarse-grained synchronization, i.e., including only a few synchronization variables to protect shared data, leading to cores highly contending for them (*high-contention*), and (ii) fine-grained synchronization, i.e., including a large number of synchronization variables, each of them protecting a small granularity of shared data, leading to cores not frequently contending for the same variables at the same time (*low-contention*). We use the term *synchronization intensity* to refer to the ratio of synchronization operations over other computation in the workload. As this ratio increases, synchronization latency affects the total execution time of the workload more.

We study three classes of applications (Table 6), all well suited for NDP. First, we evaluate pointer chasing workloads, i.e., lock-based concurrent data structures from the ASCYLIB library [31], used as key-value sets. In ASCYLIB's Binary Search Tree (BST) [37], the lock memory requests are only 0.1% of the total memory requests, so we also evaluate an external fine-grained locking BST from [130]. Data structures are initialized with a fixed size and statically partitioned across NDP units, except for BSTs, which are distributed randomly. In these benchmarks, each core performs a fixed number of operations. We use lookup operations for data structures that support it, deletion for the rest, and push and pop operations for stack and queue. Second, we evaluate graph applications with fine-grained synchronization from Crono [7, 65] (push version), where the output array has read-write data. All real-world graphs [32] used are undirected and statically partitioned across NDP units, where the vertex data is equally distributed across cores. Third, we evaluate time series analysis [142], using SCRIMP, and *real* data sets from Matrix Profile [152]. We replicate the input data in each NDP unit and partition the output array (read-write data) across NDP units.

Comparison Points. We compare *SynCron* with three schemes: (i) *Central*: a message-passing scheme that supports all primitives by extending the barrier primitive of Tesseract [8], i.e., one dedicated NDP core in the entire NDP system acts as server and coordinates synchronization among all NDP cores of the system by issuing memory requests to synchronization variables via its memory hierarchy, while the remaining client cores communicate with it via hardware message-passing; (ii) *Hier*: a hierarchical message-passing scheme that supports all primitives, similar to the barrier primitive of [43] (or hierarchical lock of [141]), i.e., one NDP core per NDP unit acts as server and coordinates synchronization by issuing memory requests to synchronization variables via its memory hierarchy (including caches), and communicates with other servers and local client cores (located at the same NDP unit with it) via hardware message-passing; (iii) *Ideal*: an ideal scheme with zero performance overhead for synchronization. In our evaluation, each NDP core runs one thread. For fair comparison, we use the same number of client cores, i.e., 15 per NDP unit, that execute the main workload for all schemes. For synchronization, we add one server core for the entire system in *Central*, one server core per NDP unit for *Hier*, and one SE per NDP unit for *SynCron*. For *SynCron*, we disable one core per NDP unit to match the same number of client cores as the previous schemes. Maintaining the same thread-level parallelism for executing the main kernel is consistent with prior works on message-passing synchronization [97, 141].

Data Structure	Configuration
Stack [31]	100K - 100% push
Queue [31, 104]	100K - 100% pop
Array Map [31, 56]	10 - 100% lookup
Priority Queue [11, 31, 118]	20K - 100% deleteMin
Skip List [31, 118]	5K - 100% deletion
Hash Table [31, 63]	1K - 100% lookup
Linked List [31, 63]	20K - 100% lookup
Binary Search Tree Fine-Grained (BST_FG) [130]	20K - 100% lookup
Binary Search Tree Drachsler (BST_Drachsler) [31, 37]	10K - 100% deletion

Real Application	Locks	Barriers
Breadth First Search (**bfs**) [7]	✓	✓
Connected Components (**cc**) [7]	✓	✓
Single Source Shortest Paths (**sssp**) [7]	✓	✓
Pagerank (**pr**) [7]	✓	✓
Teenage Followers (**tf**) [65]	✓	-
Triangle Counting (**tc**) [7]	✓	✓
Time Series Analysis (**ts**) [152]	✓	✓

Real Application	Input Data Set
bfs, cc, sssp, pr, tf, tc	wikipedia -20051105 (**wk**)
	soc-LiveJournal1 (**sl**)
	sx-stackoverflow (**sx**)
	com-Orkut (**co**)
ts	air quality (**air**)
	energy consumption (**pow**)

Table 6: Summary of all workloads used in our evaluation.

6. Evaluation

6.1. Performance

6.1.1. Synchronization Primitives. Figure 10 evaluates all supported primitives using 60 cores, varying the interval (in terms of instructions) between two synchronization points. We devise simple benchmarks, where cores repeatedly request a single synchronization variable. For lock, the critical section is empty, i.e., it does not include any instruction. For semaphore and condition variable, half of the cores execute sem_wait/cond_wait, while the rest execute sem_post/cond_signal, respectively. As the interval between synchronization points becomes smaller, *SynCron*'s performance benefit increases. For an interval of 200 instructions, *SynCron* outperforms *Central* and *Hier* by 3.05× and 1.40× respectively, averaged across all primitives. *SynCron* outperforms *Hier* due to directly buffering synchronization variables in low-latency STs, and achieves the highest benefits in the condition variable primitive (by 1.61×), since this benchmark has higher synchronization intensity compared to the rest: cores coordinate for both the condition variable and the lock associated with it. When the interval between synchronization operations becomes larger, synchronization requests become less dominant in the main workload, and thus all schemes perform similarly. Overall, *SynCron* outperforms prior schemes for all different synchronization primitives.

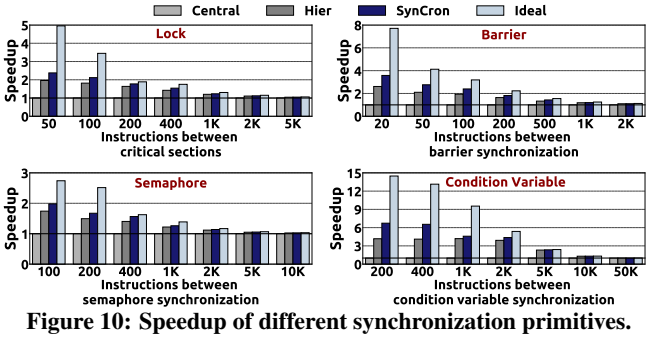


Figure 10: Speedup of different synchronization primitives.

6.1.2. Pointer Chasing Data Structures. Figure 11 shows the throughput for all schemes in pointer chasing varying the NDP cores in steps of 15, each time adding one NDP unit.

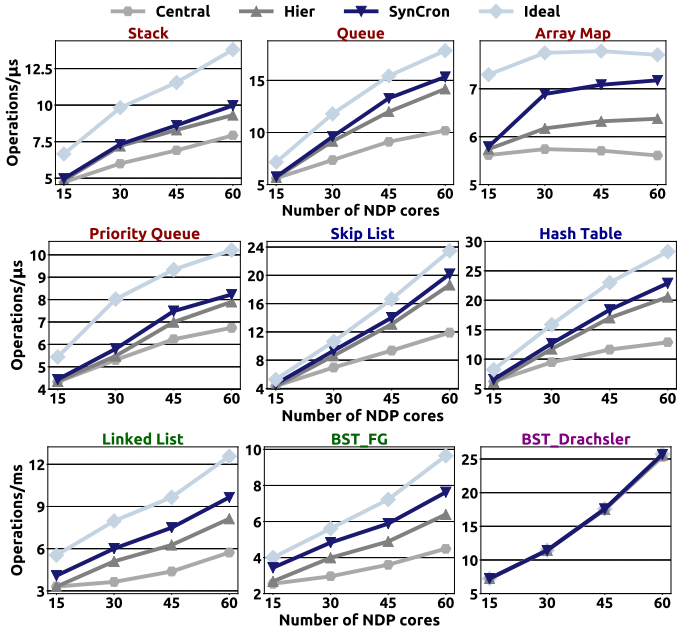


Figure 11: Throughput of pointer chasing using data structures.

We observe four different patterns. First, *stack*, *queue*, *array map*, and *priority queue* incur high contention, as all cores heavily contend for a few variables. *Array map* has the lowest scalability due to a larger critical section. In high-contention scenarios, hierarchical schemes (*Hier*, *SynCron*) perform better by reducing the expensive traffic across NDP units. *SynCron* outperforms *Hier*, since the latency cost of using SEs that update small STs is lower than using NDP cores as servers that update larger caches. Second, *skip list* and *hash table* incur medium contention, as different cores may work on different parts of the data structure. For these data structures, hierarchical schemes perform better, as they minimize the expensive traffic, and multiple server cores concurrently serve requests to their local memory. *SynCron* retains most of the performance benefits of *Ideal*, incurring only 19.9% overhead with 60 cores, and outperforms *Hier* by 9.8%. Third, *linked list* and *BST_FG* exhibit low contention and high synchronization demand, as each core requests multiple locks concurrently. These data structures cause higher synchronization-related traffic inside the network compared to *skip list* and *hash table*, and thus *SynCron* further outperforms *Hier* by 1.19× due to directly buffering synchronization variables in STs. Fourth, in *BST_Drachsler* lock requests constitute only 0.1% of the total requests, and all schemes perform similarly. Overall, we conclude that *SynCron* achieves higher throughput *than prior mechanisms* under different scenarios with diverse conditions.

6.1.3. Real Applications. Figure 12 shows the performance of all schemes with real applications using all NDP units, normalized to *Central*. Averaged across 26 application-input combinations, *SynCron* outperforms *Central* by 1.47× and *Hier* by 1.23×, and performs within 9.5% of *Ideal*.

Our real applications exhibit low contention, as two cores rarely contend for the same synchronization variable, and high synchronization demand, as several synchronization variables are active during execution. We observe that *Hier* and *SynCron* increase parallelism, because the per-NDP-unit servers service different synchronization requests concurrently, and avoid remote synchronization messages across NDP units. Even though *Hier* performs 1.19× better than *Central*, on average, its performance is still 1.33× worse than *Ideal*. *SynCron* provides most of the performance benefits of *Ideal* (with only 9.5% overhead on average), and outperforms *Hier* due to directly buffering the synchronization variables in STs, thereby completely avoiding the memory accesses for synchronization requests. Specifically, we find that *time series analysis* has high synchronization intensity, since the ratio of synchronization over other computation of the workload is higher compared to graph workloads. For this application, *Hier* and *SynCron* outperform *Central* by 1.64× and 2.22×, as they serve multiple synchronization requests concurrently. *SynCron* further outperforms *Hier* by 1.35× due to directly buffering the synchronization variables in STs. We conclude that *SynCron* performs best across *all* real application-input combinations and approaches the *Ideal* scheme with no synchronization overhead.

Scalability. Figure 13 shows the scalability of real applications using *SynCron* from 1 to 4 NDP units. Due to space limitations, we present a subset of our workloads, but we report average values for all 26 application-input combinations. This also applies for all figures presented henceforth. Across all workloads, *SynCron* enables performance scaling by at least 1.32×, on average 2.03×, and up to 3.03×, when using 4 NDP units (60 NDP cores) over 1 NDP unit (15 NDP cores).

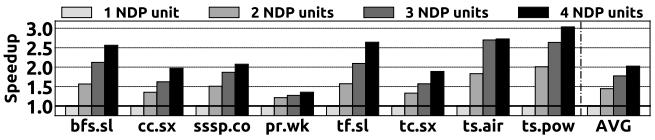


Figure 13: Scalability of real applications using *SynCron*.

6.2. Energy Consumption

Figure 14 shows the energy breakdown for cache, network, and memory in our real applications when using all cores. *SynCron* reduces the network and memory energy thanks to its hierarchical design and direct buffering. On average, *SynCron* reduces energy consumption by 2.22× over *Central* and 1.94× over *Hier*, and incurs only 6.2% energy overhead over *Ideal*.

We observe that 1) cache energy consumption constitutes a small portion of the total energy, since these applications have irregular access patterns. NDP cores that act as servers

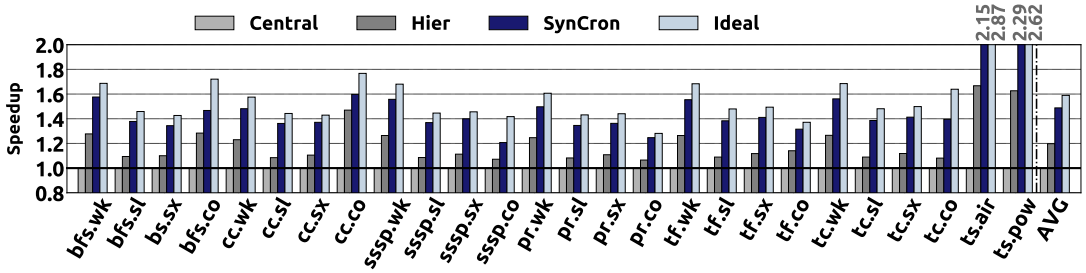


Figure 12: Speedup in real applications normalized to *Central*.

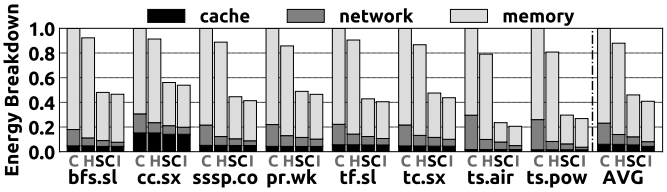


Figure 14: Energy breakdown in real applications for C: *Central*, H: *Hier*, SC: *SynCron* and I: *Ideal*.

for *Central* and *Hier* increase the cache energy only by 5.1% and 4.8% over *Ideal*. 2) *Central* generates a larger amount of expensive traffic across NDP units compared to hierarchical schemes, resulting in 2.68× higher network energy over *SynCron*. *SynCron* also has less network energy (1.21×) than *Hier*, because it avoids transferring synchronization variables from memory to SEs due to directly buffering them. 3) *Hier* and *Central* have approximately the same memory energy consumption, because they issue a similar number of requests to memory. In contrast, *SynCron*'s memory energy consumption is similar to that of *Ideal*. We note that *SynCron* provides *higher* energy reductions in applications with high synchronization intensity, such as time series analysis, since it avoids a *higher* number of memory accesses for synchronization due to its direct buffering capability.

6.3. Data Movement

Figure 15 shows normalized data movement, i.e., bytes transferred between NDP cores and memory, for all schemes using four NDP units. *SynCron* reduces data movement across all workloads by 2.08× and 2.04× over *Central* and *Hier*, respectively, on average, and incurs only 13.8% more data movement than *Ideal*. *Central* generates high data movement across NDP units, particularly when running time series analysis that has high synchronization intensity. *Hier* reduces the traffic across NDP units; however, it may increase the traffic inside an NDP unit, occasionally leading to slightly higher total data movement (e.g., *ts.air*). This is because when an NDP core requests a synchronization variable that is physically located in another NDP unit, it first sends a message inside the NDP unit to its local server, which in turns sends a message to the global server. In contrast, *SynCron* reduces the traffic inside an NDP unit due to directly buffering synchronization variables, and across NDP units due to its hierarchical design.

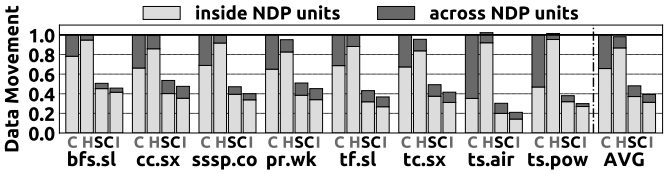


Figure 15: Data movement in real applications for C: *Central*, H: *Hier*, SC: *SynCron* and I: *Ideal*.

6.4. Non-Uniformity of NDP Systems

6.4.1. High Contention. Hierarchical schemes provide high benefit under high contention, as they prioritize local requests inside each NDP unit. We study their performance benefit in stack and priority queue (Figure 16) when varying the transfer latency of the interconnection links used across four NDP units. *Central* is significantly affected by the interconnect latency across NDP units, as it is oblivious to the non-uniform nature of the NDP system. Observing *Ideal*, which reflects the actual behavior of the main workload, we notice that after a certain point (vertical line), the cost of remote memory accesses across NDP units become high enough to dominate performance. *SynCron* and *Hier* tend to follow the actual behavior of the workload, as local synchronization messages within NDP units are much less expensive than remote messages of *Central*. *SynCron* outperforms *Hier* by 1.06× and 1.04× for stack and priority queue. We conclude that *SynCron* is the best at hiding the latency of slow links across NDP units.

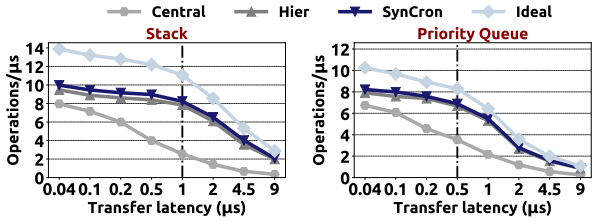


Figure 16: Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units.

6.4.2. Low Contention. We also study the effect of interconnection links used across the NDP units in a low-contention graph application (Figure 17). Observing *Ideal*, with 500 ns transfer latency per cache line, we note that the workload experiences 2.46× slowdown over the default latency of 40 ns, as 24.1% of its memory accesses are to remote NDP units. As the transfer latency increases, *Central* incurs significant slowdown over *Ideal*, since all NDP cores of the system communicate with one single server, generating expensive traffic across NDP units. In contrast, the slowdown of hierarchical schemes over *Ideal* is smaller, as these schemes generate less remote traffic by distributing the synchronization requests across multiple local servers. *SynCron* outperforms *Hier* due to its direct buffering capabilities. Overall, *SynCron* outperforms prior high-performance schemes even when the network delay across NDP units is large.

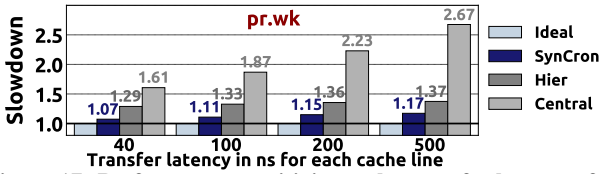


Figure 17: Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units. All data is normalized to *Ideal* (*lower is better*).

6.5. Memory Technologies

We study three memory technologies, which provide different memory access latencies and bandwidth. We evaluate (i) 2.5D NDP using HBM, (ii) 3D NDP using HMC, and (iii) 2D NDP using DDR4. Figure 18 shows the performance of all schemes normalized to *Central* of each memory. The reported values show the speedup of *SynCron* over *Central* and *Hier*. *SynCron*'s benefit is independent of the memory used: its performance versus *Ideal* only slightly varies (±1.4%) across different memory technologies, since STs never overflow. Moreover, *SynCron*'s performance improvement over prior schemes increases as the memory access latency becomes higher thanks to direct buffering, which avoids expensive memory accesses for synchronization. For example, in *ts.pow*, *SynCron* outperforms *Hier* by 1.41× and 2.49× with HBM and DDR4, respectively, as the latter incurs higher access latency. Overall, *SynCron* is orthogonal to the memory technology used.

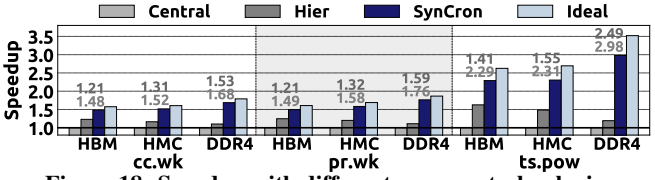
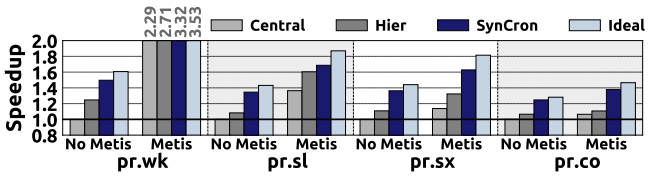


Figure 18: Speedup with different memory technologies.

6.6. Effect of Data Placement

Figure 19 evaluates the effect of better data placement on *SynCron*'s benefits. We use Metis [74] to obtain a 4-way graph partitioning to minimize the crossing edges between the 4 NDP units. All data values are normalized to *Central* without Metis. For *SynCron*, we define ST occupancy as the average fraction of ST entries that are occupied in each cycle.



Max ST Occupancy (%)	pr.wk	pr.sl	pr.sx	pr.co
No Metis	62	51	53	48
Metis	39	29	38	34

Figure 19: Performance sensitivity to a better graph partitioning and maximum ST occupancy of *SynCron*.

We make three observations. First, *Ideal*, which reflects the actual behavior of the main kernel (i.e., with zero synchronization overhead), improves performance by 1.47× across the four graphs. Second, with a better graph partitioning, *SynCron* still outperforms both *Central* and *Hier*. Third, we find that ST occupancy is lower with a better graph partitioning. When a local SE receives a request for a synchronization variable of another NDP unit, *both* the local SE and the *Master SE* reserve a new entry in their STs. With a better graph partitioning, NDP cores send requests to their local SE, which is also the *Master SE* for the requested variable. Thus, *only one* SE of the system reserves a new entry, resulting in a lower ST occupancy. We conclude that, with better data placement *SynCron* still performs the best while achieving even lower ST occupancy.

6.7. *SynCron*'s Design Choices

6.7.1. Hierarchical Design. To demonstrate the effectiveness of *SynCron*'s hierarchical design in non-uniform NDP systems, we compare it with *SynCron*'s *flat* variant. Each core in *flat directly* sends all its synchronization requests to the *Master SE* of each variable. In contrast, each core in *SynCron* sends all its synchronization requests to the local SE. If the local SE is *not* the *Master SE* for the requested variable, the local SE sends a message across NDP units to the *Master SE*.

We evaluate three synchronization scenarios: (i) low-contention and synchronization non-intensive (e.g., graph applications), (ii) low-contention and synchronization-intensive (e.g., time series analysis), and (iii) high-contention (e.g., queue data structure).

Low-contention and synchronization non-intensive. Figure 20 evaluates this scenario using several graph processing workloads with 40 ns link latency between NDP units. *SynCron* is 1.1% worse than *flat*, on average. We conclude that *SynCron* performs only *slightly* worse than *flat* for low-contention and synchronization non-intensive scenarios.

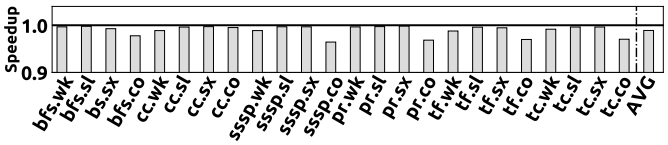


Figure 20: Speedup of *SynCron* normalized to *flat* with 40 ns link latency between NDP units, under a low-contention and synchronization non-intensive scenario.

Low-contention and synchronization-intensive. Figure 21a evaluates this scenario using time series analysis with four different link latency values between NDP units. *SynCron* performs 7.3% worse than *flat* with a 40 ns inter-NDP-unit latency. With a 500 ns inter-NDP-unit latency, *SynCron* performs *only* 3.6% worse than *flat*, since remote traffic has a larger impact on the total execution time. We conclude that *SynCron* performs modestly worse than *flat*, and *SynCron*'s slowdown decreases as non-uniformity, i.e., the latency between NDP units, increases.

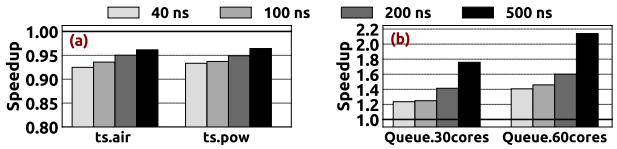


Figure 21: Speedup of *SynCron* normalized to *flat*, as we vary the transfer latency of the interconnection links used to connect NDP units, under (a) a low-contention and synchronization-intensive scenario using 4 NDP units, and (b) a high-contention scenario using 2 and 4 NDP units.

High-contention. Figure 21b evaluates this scenario using a queue data structure with four different link latency values between NDP units, for 30 and 60 NDP cores. *SynCron* with 30 NDP cores outperforms *flat* from 1.23× to 1.76×, as the inter-NDP-unit latency increases from 40 ns to 500 ns (i.e., with increasing non-uniformity in the system). In a scenario with high non-uniformity in the system and large number of contended cores, e.g., using a 500 ns inter-NDP-unit latency and 60 NDP cores, *SynCron*'s benefit increases to a 2.14× speedup over *flat*. We conclude that *SynCron* performs *significantly* better than *flat* under high-contention.

Overall, we conclude that in *non-uniform*, *distributed* NDP systems, *only* a *hierarchical* hardware synchronization design can achieve high performance under *all* various scenarios.

6.7.2. ST Size. We show the effectiveness of the proposed 64-entry ST (per NDP unit) using real applications. Table 7 shows the measured occupancy across all STs. Figure 22 shows the performance sensitivity to ST size. In graph applications, the average ST occupancy is low (2.8%), and the 64-entry ST never overflows: maximum occupancy is 63% (*cc.wk*). In contrast, time series analysis has higher ST occupancy (reaching up to 89% in *ts.pow*) due to the high synchronization intensity, but there are no ST overflows. Even a 48-entry ST overflows for only 0.01% of synchronization requests, and incurs 2.1% slowdown over a 64-entry ST. We conclude that the proposed 64-entry ST meets the needs of applications that have high synchronization intensity.

ST Occupancy	Max (%)	Avg (%)
bfs.wk	51	1.33
bfs.sl	59	1.49
bfs.sx	51	3.24
bfs.co	55	6.09
cc.wk	63	1.27
cc.sl	61	2.16
cc.sx	48	2.43
cc.co	46	4.53
sssp.wk	62	1.18
sssp.sl	54	2.08
sssp.sx	50	2.20
sssp.co	48	5.23
pr.wk	62	4.27
pr.sl	51	2.27
pr.sx	53	2.46
pr.co	48	4.72
tf.wk	62	1.44
tf.sl	53	2.21
tf.sx	50	2.99
tf.co	48	4.61
tc.wk	62	1.26
tc.sl	48	2.08
tc.sx	50	2.77
tc.co	51	4.52
ts.air	84	44.20
ts.pow	89	43.51

Table 7: ST occupancy in real applications.

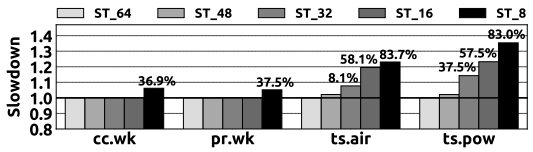


Figure 22: Slowdown with varying ST size (normalized to 64-entry ST). Numbers on top of bars show the percentage of overflowed requests.

6.7.3. Overflow Management. The linked list and BST_FG data structures are the *only* cases where the proposed 64-entry ST overflows, when using 60 cores, for 3.1% and 30.5% of the requests, respectively. This is because each core requests at least two locks *at the same time* during the execution. Note that these synthetic benchmarks represent extreme scenarios, where all cores repeatedly perform key-value operations.

Figure 23 compares BST_FG's performance with *SynCron*'s integrated overflow scheme versus with a non-integrated scheme as in MiSAR. When overflow occurs, MiSAR's accelerator aborts all participating cores notifying them to use an alternative synchronization library, and when the cores finish synchronizing via an alternative solution, they notify MiSAR's accelerator to switch back to hardware synchronization. We adapt this scheme to *SynCron* for comparison purposes: when an ST overflows, SEs send abort messages to NDP cores with a hierarchical protocol, notifying them to use an alternative synchronization solution, and after finishing synchronization they notify SEs to decrease their indexing counters and switch to hardware. We evaluate two alternative solutions: (i) *SynCron_CentralOvrfl*, where one dedicated NDP core handles all synchronization variables, and (ii) *SynCron_DistribOvrfl*, where one NDP core per NDP unit handles variables located in the same NDP unit. With 30.5% overflowed requests (i.e., with a 64-entry ST), *SynCron_CentralOvrfl* and *SynCron_DistribOvrfl* incur 12.3% and 10.4% performance slowdown compared to with *no* ST overflow, due to high network traffic and communication costs between NDP cores and SEs. In contrast, *SynCron* affects performance by only 3.2% compared to with *no* ST overflow. We conclude that *SynCron*'s integrated hardware-only overflow scheme enables very small performance overhead.

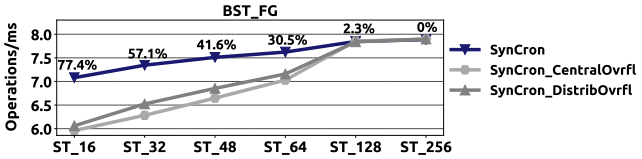


Figure 23: Throughput achieved by BST_FG using different overflow schemes and varying the ST size. The reported numbers show to the percentage of overflowed requests.

6.8. *SynCron*'s Area and Power Overhead

Table 8 compares an SE with the ARM Cortex A7 core [14]. We estimate the SPU using Aladdin [129], and the ST and indexing counters using CACTI [109]. We conclude that our proposed hardware unit incurs very modest area and power costs to be integrated into the compute die of an NDP unit.

	SE (Synchronization Engine)	ARM Cortex A7 [14]
Technology	40nm	28nm
Area	SPU: 0.0141mm^2, ST: 0.0112mm^2 Indexing Counters: 0.0208mm^2 **Total:** 0.0461mm^2	32KB L1 Cache **Total:** 0.45mm^2
Power	2.7 mW	100mW

Table 8: Comparison of SE with a simple general-purpose in-order core, ARM Cortex A7.

7. Related Work

To our knowledge, our work is the first one to (i) comprehensively analyze and evaluate synchronization primitives in NDP systems, and (ii) propose an end-to-end hardware-based synchronization mechanism for efficient execution of such primitives. We briefly discuss prior work.

Synchronization on NDP. Ahn et al. [8] include a message-passing barrier similar to our *Central* baseline. Gao et al. [43] implement a hierarchical tree-based barrier for HMC [59], where cores first synchronize inside the vault, then across vaults, and finally across HMC stacks. Section 6.1 shows that *SynCron* outperforms such schemes. Gao et al. [43] also provide remote atomics at the vault controllers of HMC. However, synchronization using remote atomics creates high global traffic and hotspots [41, 96, 108, 147, 153].

Synchronization on CPUs. A range of hardware synchronization mechanisms have been proposed for commodity CPU systems [1–3, 10, 116, 124]. These are not suitable for NDP systems because they either (i) rely on the underlying cache coherence system [10, 124], (ii) are tailored for the 2D-mesh network topology to connect all cores [2, 3], or (iii) use transmission-line technology [116] or on-chip wireless technology [1]. Callbacks [120] includes a directory cache structure close to the LLC of a CPU system built on self-invalidation coherence protocols [26, 75, 77, 91, 121, 139]. Although it has low area cost, it would be oblivious to the non-uniformity of NDP, thereby incurring high performance overheads under high contention (Section 6.7.1). Callbacks improves performance of spin-wait in hardware, on top of which high-level primitives (locks/barriers) are implemented in software. In contrast, *SynCron* directly supports high-level primitives in hardware, and is tailored to all salient characteristics of NDP systems.

The closest works to ours are SSB [157], LCU [146], and MiSAR [97]. SSB, a shared memory scheme, includes a small buffer attached to each controller of LLC to provide lock semantics for a given data address. LCU, a message-passing scheme, incorporates a control unit into each core and a reservation table into each memory controller to provide reader-writer locks. MiSAR is a message-passing synchronization accelerator distributed at each LLC slice of tile-based many-core chips. These schemes provide efficient synchronization for CPU systems *without* relying on hardware coherence protocols. As shown in Table 4, compared to these works, *SynCron* is a more effective, general and easy-to-use solution for NDP systems. These works have two major shortcomings. First, they are designed for *uniform* architectures, and would incur high performance overheads in *non-uniform, distributed* NDP systems under high-contetion scenarios, similarly to *flat* in Figure 21b. Second, SSB and LCU handle overflow cases using software exception handlers that typically incur large performance overheads, while MiSAR's overflow scheme would incur high performance degradation due to high network traffic and communication costs between the cores and the synchronization accelerator (Section 6.7.3). In contrast, *SynCron* is a non-uniformity aware, hardware-only, end-to-end solution designed to handle key characteristics of NDP systems.

Synchronization on GPUs. GPUs support remote atomic units at the shared cache and hardware barriers among threads of the same block [114], while inter-block barrier synchronization is inefficiently implemented via the host CPU [114]. The closest work to ours is HQL [153], which modifies the tag arrays of L1 and L2 caches to support the lock primitive. This scheme incurs high area cost [41], and is tailored to the GPU architecture that includes a shared L2 cache, while most NDP systems do *not* have shared caches.

Synchronization on MPPs. The Cray T3D/T3E [81, 127], SGI Origin [88], and AMOs [154] include remote atomics at the memory controller, while NYU Ultracomputer [52] provides *fetch&and* remote atomics in each network switch. As discussed in Section 2, synchronization via remote atomics incurs high performance overheads due to high global traffic [41, 108, 147, 153]. Cray T3E supports a barrier using physical wires, but it is designed specifically for 3D torus interconnect. Tera MTA [12], HEP [71, 134], J- and M-machines [29, 78], and Alewife [5] provide synchronization

using hardware bits (*full/empty* bits) as tags in *each memory word*. This scheme can incur high area cost [146]. QOLB [51] associates one cache line for every lock to track a pointer to the next waiting core, and one cache line for local spinning using bits (*syncbits*). QOLB is built on the underlying cache coherence protocol. Similarly, DASH [95] keeps a queue of waiting cores for a lock in the directory used for coherence to notify caches when the lock is released. CM5 [94] supports remote atomics and a barrier among cores via a dedicated physical control network (organized as a binary tree), which would incur high hardware cost to be supported in NDP systems.

8. Conclusion

SynCron is the first end-to-end synchronization solution for NDP systems. *SynCron* avoids the need for complex coherence protocols and expensive *rmw* operations, incurs very modest hardware cost, generally supports many synchronization primitives and is easy-to-use. Our evaluations show that it outperforms prior designs under various conditions, providing high performance both under high-contention (due to reduction of expensive traffic across NDP units) and low-contention scenarios (due to direct buffering of synchronization variables and high execution parallelism). We conclude that *SynCron* is an efficient synchronization mechanism for NDP systems, and hope that this work encourages further comprehensive studies of the synchronization problem in heterogeneous systems, including NDP systems.

Acknowledgments

We thank the anonymous reviewers of ISCA 2020, MICRO 2020 and HPCA 2021 for feedback. We thank Dionisios Pnevmatikatos, Konstantinos Nikas, Athena Elafrou, Foteini Strati, Dimitrios Siakavaras, Thomas Lagos, Andreas Triantafyllos for helpful technical discussions. We acknowledge support from the SAFARI group's industrial partners, especially ASML, Google, Facebook, Huawei, Intel, Microsoft, VMware, and Semiconductor Research Corporation. During part of this research, Christina Giannoula was funded from the General Secretariat for Research and Technology (GSRT) and the Hellenic Foundation for Research and Innovation (HFRI).

References

[1] S. Abadal, A. Cabellos-Aparicio, E. Alarcon, and J. Torrellas, "WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication," in *ASPLOS*, 2016.
[2] J. L. Abellán, J. Fernández, and M. E. Acacio, "A g-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs," in *ICPP*, 2010.
[3] J. L. Abellán, J. Fernández, M. E. Acacio *et al.*, "Glocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs," in *IPDPS*, 2011.
[4] M. Abeydeera and D. Sanchez, "Chronos: Efficient Speculative Parallelism for Accelerators," in *ASPLOS*, 2020.
[5] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson *et al.*, "The MIT Alewife Machine: Architecture and Performance," in *ISCA*, 1998.
[6] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed on Chip Network Model inside a Full-System Simulator," in *ISPASS*, 2009.
[7] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *IISWC*, 2015.
[8] J. Ahn, S. Hong, S. Yoo, and O. Mutlu, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[9] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[10] B. S. Akgul, J. Lee, and V. J. Mooney, "A System-on-a-Chip Lock Cache with Task Preemption Support," in *CASES*, 2001.
[11] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A Scalable Relaxed Priority Queue," in *PPoPP*, 2015.
[12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz *et al.*, "The Tera Computer System," *ICS*, 1990.
[13] T. Anderson, "The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors," in *ICPP*, 1989.
[14] ARM, "Cortex-A7 Technical Reference Manual," 2009.
[15] A. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server," in *BDCC*, 2015.
[16] A. J. Awan, V. Vlassov, M. Brorsson, and E. Ayguade, "Node Architecture Implications for In-Memory Data Analytics on Scale-in Clusters," in *BDCAT*, 2016.
[17] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, 2014.
[18] U. N. Bhat, *An Introduction to Queueing Theory: Modeling and Analysis in Applications*, 2nd ed. Birkhäuser Basel, 2015.
[19] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[20] A. Boroumand, S. Ghose, M. Patel, H. Hassan *et al.*, "CoNDA: Efficient Cache Coherence Support for Near-data Accelerators," in *ISCA*, 2019.
[21] A. Boroumand, S. Ghose, M. Patel, H. Hassan *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *CAL*, 2017.
[22] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev *et al.*, "An Analysis of Linux Scalability to Many Cores," in *OSDI*, 2010.
[23] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.
[24] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High Performance Locks for Multi-Level NUMA Systems," *PPoPP*, 2015.
[25] J. Choe, A. Huang, T. Moreshet, M. Herlihy *et al.*, "Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation," in *SPAA*, 2019.
[26] B. Choi, R. Komuravelli, H. Sung, R. Smolinski *et al.*, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011.
[27] T. Craig, "Building FIFO and Priority Queuing Spin Locks from Atomic Swap," Tech. Rep., 1993.
[28] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware-Software Approach*, 1999.
[29] W. Dally, J. S. Fiske, J. Keen, R. Lethin *et al.*, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, 1992.
[30] T. David, R. Guerraoui, and . V. Trigonakis, "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask," in *SOSP*, 2013.
[31] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," in *ASPLOS*, 2015.
[32] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *TOMS*, 2011.
[33] G. F. de Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose *et al.*, "A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study," in *SIGMETRICS*, 2021.
[34] F. Devaux, "The True Processing in Memory Accelerator," in *Hot Chips*, 2019.
[35] D. Dice, V. J. Marathe, and N. Shavit, "Flat-Combining NUMA Locks," in *SPAA*, 2011.
[36] D. Dice, V. J. Marathe, and N. Shavit, "Lock Cohorting: A General Technique for Designing NUMA Locks," *TOPC*, 2015.
[37] D. Drachsler, M. Vechev, and E. Yahav, "Practical Concurrent Binary Search Trees via Logical Ordering," *PPoPP*, 2014.
[38] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.
[39] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[40] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-Free Symmetric Sparse Matrix-Vector Multiplication on Multicore Architectures," in *SC*, 2019.
[41] A. ElTantawy and T. M. Aamodt, "Warp Scheduling for Fine-Grained Synchronization," in *HPCA*, 2018.
[42] I. Fernandez, R. Quislant, C. Giannoula, M. Alser *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," *ICCD*, 2020.
[43] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[44] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[45] M. Gao, J. Pu, X. Yang, M. Horowitz *et al.*, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[46] S. Ghose, A. Boroumand, J. Kim, J. Gómez-Luna *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.
[47] S. Ghose, T. Li, N. Hajinazar, D. Senol Cali *et al.*, "Demystifying Complex Workload-DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.
[48] C. Giannoula, G. Goumas, and N. Koziris, "Combining HTM with RCU to Speed up Graph Coloring on Multicore Platforms," in *ISC HPC*, 2018.
[49] M. Gokhale, S. Lloyd, and C. Hajas, "Near Memory Data Structure Rearrangement," in *MEMSYS*, 2015.
[50] J. Gomez-Luna, I. El Hajj, I. Fernandez, C. Giannoula *et al.*, "Benchmarking a New Paradigm: Understanding a Modern Processing-in-Memory Architecture," in *SIGMETRICS*, 2021.
[51] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," in *ASPLOS*, 1989.
[52] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe *et al.*, "The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine," in *ISCA*, 1982.
[53] D. Grunwald and S. Vajracharya, "Efficient Barriers for Distributed Shared Memory Computers," in *IPDPS*, 1994.
[54] P. Gu, S. Li, D. Stow, R. Barnes *et al.*, "Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges," in *GLSVLSI*, 2016.
[55] P. Gu, X. Xie, Y. Ding, G. Chen *et al.*, "IPIM: Programmable in-Memory Image Processing Accelerator Using Near-Bank Architecture," *ISCA*, 2020.
[56] R. Guerraoui and V. Trigonakis, "Optimistic Concurrency with OPTIK," PPoPP 2016.
[57] H. Guiroux, R. Lachaize, and V. Quéma, "Multicore Locks: The Case Is Not Closed Yet," in *USENIX ATC*, 2016.
[58] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides Benítez, and N. Guil Mata, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," *TPDS*, 2013.
[59] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay *et al.*, "Demystifying the Characteristics of 3D-stacked Memories: A case Study for Hybrid Memory Cube," in *IISWC*, 2017.
[60] M. Hashemi, E. Ebrahimi, O. Mutlu, Y. N. Patt *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[61] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta, "A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," *TC*, 1999.
[62] D. Hensgen, R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming*, 1988.
[63] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2008.
[64] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "A Survey of Barrier Algorithms for Coarse Grained Supercomputers," *Chemnitzer Informatik Berichte*, 2004.
[65] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying Scalable Graph Processing with a Domain-Specific Language," in *CGO*, 2014.
[66] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee *et al.*, "Transparent Offloading and Mapping: Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[67] K. Hsieh, S. Khan, N. Vijaykumar, K. Chang *et al.*, "Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[68] Intel, *64 and IA-32 Architectures Software Developer's Manual*, 2009.
[69] J. Joao, M. A. Suleman, O. Mutlu, and Y. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.
[70] J. Joao, M. A. Suleman, O. Mutlu, and Y. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *ISCA*, 2013.
[71] H. F. Jordan, "Performance Measurements on HEP - a Pipelined MIMD Computer," *ISCA*, 1983.
[72] H. Jun, J. Cho, K. Lee, H.-Y. Son *et al.*, "HBM DRAM Technology and Architecture," in *IMW*, 2017.
[73] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi *et al.*, "SMASH: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations," in *MICRO*, 2019.
[74] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, 1998.
[75] S. Kaxiras and G. Keramidas, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power," *IEEE Micro*, 2010.
[76] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros *et al.*, "Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory," in *HPDC*, 2015.
[77] S. Kaxiras and A. Ros, "A New Perspective for Efficient Virtual-Cache Coherence," in *ISCA*, 2013.
[78] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter *et al.*, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," in *ISCA*, 1998.
[79] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta *et al.*, "Cohesion: A Hybrid Memory Model for Accelerators," in *ISCA*, 2010.
[80] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAYPOINT: Scaling Coherence to Thousand-Core Architectures," in *PACT*, 2010.
[81] R. E. Kessler and J. L. Schwarzmeier, "Cray T3D: A New Dimension for Cray Research," *Digest of Papers. Compcon Spring*, 1993.
[82] D. Kim, J. Kung, S. Chai, S. Yalamanchili *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[83] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *PACT*, 2013.
[84] J. Kim, D. Senol Cali, H. Xin, D. Lee *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
[85] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015. https://github.com/CMU-SAFARI/ramulator
[86] Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *TC*, 1979.
[87] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Commun. ACM*, 1974.
[88] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *ISCA*, 1997.
[89] D. Lavenier, J.-F. Roy, and D. Furodet, "DNA Mapping using Processor-in-Memory Architecture," in *BIBM*, 2016.
[90] M. LeBeane, S. Song, R. Panda, J. H. Ryoo *et al.*, "Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters," in *SC*, 2015.
[91] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," *ISCA*, 1995.
[92] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim *et al.*, "25.2 A 1.2V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV," in *ISSCC*, 2014.
[93] D. Lee, S. Ghose, G. Pekhimenko, S. Khan *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
[94] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman *et al.*, "The Network Architecture of the Connection Machine CM-5 (Extended Abstract)," in *SPAA*, 1992.
[95] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber *et al.*, "The Stanford Dash Multiprocessor," *Computer*, 1992.
[96] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-Grained Synchronizations and Dataflow Programming on GPUs," in *ICS*, 2015.
[97] C. Liang and M. Prvulovic, "MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management," in *ISCA*, 2015.
[98] J. Liu, H. Zhao, M. A. Ogleari, D. Li *et al.*, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.
[99] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.
[100] V. Luchangco, D. Nussbaum, and N. Shavit, "A Hierarchical CLH Queue Lock," in *Euro-Par*, 2006.
[101] P. Magnusson, A. Landin, and E. Hagersten, "Queue Locks on Cache Coherent Multiprocessors," in *IPDPS*, 1994.
[102] J. Mellor-Crummey and M. Scott, "Synchronization without Contention," *ASPLOS*, 1991.
[103] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *TOCS*, 1991.
[104] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *PODC*, 1996.
[105] A. Mirhosseini and J. Torrellas, "Survive: Pointer-Based In-DRAM Incremental Check-Pointing for Low-Cost Data Persistence and Rollback-Recovery," *CAL*, 2016.
[106] S. A. Mojumder, M. S. Louis, Y. Sun, A. K. Ziabari *et al.*, "Profiling DNN Workloads on a Volta-based DGX-1 System," in *IISWC*, 2018.
[107] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *PACT*, 2009.
[108] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates," in *MICRO*, 2019.
[109] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[110] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
[111] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MICPRO*, 2019.
[112] L. Nai, R. Hadidi, J. Sim, H. Kim *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[113] R. Nair, S. F. Antao, C. Bertolli, P. Bose *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM JRD*, 2015.
[114] NVIDIA, "NVIDIA Tesla V100 GPU Architecture," *White Paper*, 2017.
[115] NVIDIA, "ONTAP AI–NVIDIA DGX-2 POD with NetApp AFF A800," *White Paper*, 2019.
[116] J. Oh, M. Prvulovic, and A. Zajic, "TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines," in *ISCA*, 2011.
[117] A. Pattnaik, X. Tang, A. Jog, O. Kayiran *et al.*, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.
[118] W. Pugh, "Concurrent Maintenance of Skip Lists," Tech. Rep., 1990.
[119] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian *et al.*, "NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
[120] A. Ros and S. Kaxiras, "Callback: Efficient Synchronization without Invalidation with a Directory just for Spin-Waiting," in *ISCA*, 2015.
[121] A. Ros and S. Kaxiras, "Complexity-Effective Multicore Coherence," PACT 2012.
[122] L. Rudolph and Z. Segall, *Dynamic Decentralized Cache Schemes for MIMD Parallel Processors*, 1984.
[123] J. Rutgers, M. Bekooij, and G. Smit, "Portable Memory Consistency for Software Managed Distributed Memory in Many-Core SoC," in *IPDPSW*, 2013.
[124] J. Sampson, R. Gonzalez, J.-F. Collard, N. Jouppi *et al.*, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *MICRO*, 2006.
[125] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA*, 2013.
[126] M. L. Scott, "Non-Blocking Timeout in Scalable Queue-based Spin Locks," in *PODC*, 2002.
[127] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *ASPLOS*, 1996.
[128] V. Seshadri, D. Lee, T. Mullins, H. Hassan *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[129] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei *et al.*, "Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin," in *MICRO*, 2016.
[130] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees," PACT 2017. https://github.com/jimsiak/concurrent-maps
[131] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
[132] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira *et al.*, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.
[133] G. Singh, L. Chelini, S. Corda, A. J. Awan *et al.*, "Near-Memory Computing: Past, Present, and Future," *MICPRO*, 2019.
[134] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," *ICPP*, 1978.
[135] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.
[136] F. Strati, C. Giannoula, D. Siakavaras, G. Goumas *et al.*, "An Adaptive Concurrent Priority Queue for NUMA Architectures," in *CF*, 2019.
[137] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib *et al.*, "Data Marshaling for Multi-Core Architectures," in *ISCA*, 2010.
[138] M. A. Suleman, O. Mutlu, M. Qureshi, and Y. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in *ASPLOS*, 2009.
[139] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism," *ASPLOS*, 2013.
[140] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing Lock Contention in Multithreaded Applications," in *PPoPP*, 2010.
[141] X. Tang, J. Zhai, X. Qian, and W. Chen, "pLock: A Fast Lock for Architectures with Explicit Inter-core Message Passing," in *ASPLOS*, 2019.
[142] S. Torkamani and V. Lohweg, "Survey on Time Series Motif Discovery," *Wiley Interdis. Rev.: Data Mining and Knowledge Discovery*, 2017.
[143] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *MICRO*, 2018.
[144] UPMEM, "https://www.upmem.com/."
[145] hybridmemorycube.org, "Hybrid Memory Cube Specification rev. 2.1," *Hybrid Memory Cube Consortium*, 2015.
[146] E. Vallejo, R. Beivide, A. Cristal, T. Harris *et al.*, "Architectural Support for Fair Reader-Writer Locking," in *MICRO*, 2010.
[147] K. Wang, D. Fussell, and C. Lin, "Fast Fine-Grained Global Synchronization on GPUs," in *ASPLOS*, 2019.
[148] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, 2011.
[149] P. T. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. E. Becker *et al.*, "Energy Model of Networks-on-Chip and a Bus," in *SOCC*, 2005.
[150] Xilinx, "Virtex UltraScale+ HBM FPGA," 2019.
[151] M. Yan, X. Hu, S. Li, A. Basak *et al.*, "Alleviating Irregularity in Graph Analytics Acceleration: A Hardware/Software Co-Design Approach," in *MICRO*, 2019.
[152] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum *et al.*, "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets," in *ICDM*, 2016.
[153] A. Yilmazer and D. Kaeli, "HQL: A Scalable Synchronization Mechanism for GPUs," in *IPDPS*, 2013.
[154] L. Zhang, Z. Fang, and J. B. Carter, "Highly Efficient Synchronization based on Active Memory Operations," in *IPDPS*, 2004.
[155] M. Zhang, Y. Zhuo, C. Wang, M. Gao *et al.*, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.
[156] M. Zhang, H. Chen, L. Cheng, F. C. M. Lau *et al.*, "Scalable Adaptive NUMA-Aware Lock," *TPDS*, 2017.
[157] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-Core Architectures," in *ISCA*, 2007.
[158] Y. Zhuo, C. Wang, M. Zhang, R. Wang *et al.*, "GraphQ: Scalable PIM-Based Graph Processing," in *MICRO*, 2019.