Wait-free Programming for General Purpose Computations on Graphics Processors

Phuong Hoai Ha, *Member, IEEE Computer Society,* Philippas Tsigas, and Otto J. Anshus, *Member, IEEE Computer Society,*

Abstract—The fact that graphics processors (GPUs) are today's most powerful computational hardware for the dollar has motivated researchers to utilize the ubiquitous and powerful GPUs for general-purpose computing. However, unlike CPUs, GPUs are optimized for processing 3D graphics (e.g. graphics rendering), a kind of data-parallel applications, and consequently, several GPUs do not support strong synchronization primitives to coordinate their cores. This prevents the GPUs from being deployed more widely for general-purpose computing.

This paper aims at bridging the gap between the lack of strong synchronization primitives in the GPUs and the need for strong synchronization mechanisms in parallel applications. Based on the intrinsic features of typical GPU architectures, we construct strong synchronization objects such as wait-free and t-resilient *read-modify-write* objects for a general model of GPU architectures without hardware synchronization primitives such as *test-and-set* and *compare-and-swap*. Accesses to the wait-free objects have time complexity $O(N)$, where N is the number of processes. The wait-free objects have the optimal space complexity $O(N^2)$. Our result demonstrates that it is possible to construct wait-free synchronization mechanisms for GPUs without strong synchronization primitives in hardware and that wait-free programming is possible for such GPUs.

Index Terms—Concurrent programming, fault-tolerance, GPGPU, interprocess synchronization, multicore computing.

1 Introduction

Graphics processors (GPUs) are emerging as powerful computational co-processors for general purpose computations. The demands of graphics as well as non-graphics applications have driven GPUs to be today's most powerful computational hardware for the dollar [1]. Moreover, unlike previous GPU architectures, which are single-instruction multiple-data (SIMD), recent GPU architectures (e.g. OpenCL architecture [2] and Compute Unified Device Architecture (CUDA) [3]) are single-program multiple-data (SPMD). The latter consists of multiple SIMD multiprocessors of which each can simultaneously execute a different instruction. This extends the set of general-purpose applications on GPUs, which are no longer restricted to follow the SIMD-programming model.

However, unlike graphics computation, general-purpose computation usually needs support for reliability and inter-process synchronization. Errors in computation domains such as radiology in which GPUs are used for medical image processing are very costly and potentially harmful to people. Although hardware errors in logic have not happened frequently, such errors are expected to become significant within the next five years due to the scaling of CMOS technology [4]. Realizing the problem, researchers have recently proposed a hardware redundancy and recovery mechanism for reliable computation on GPUs [5].

In this paper, we explore the possibilities of addressing the GPU reliability issues, namely crash failures, at the software layer. Particularly we are looking at fault-tolerant synchronization techniques such as non-blocking and wait-free programming [6]. Recently, *blocking* synchronization mechanisms to synchronize threads running on *different* cores of a GPU (namely, global barriers) have been reported [7], [8]. However, unlike the computation using *non-blocking* synchronization, the computation using the traditional *blocking* synchronization (e.g. barrier and mutual exclusion) is vulnerable to deadlock caused by both scientists inexperience and scheduling mechanisms. It is notoriously difficult for scientists to deal with the deadlock when their computation needs to block many threads. *Non-preemptive* scheduling mechanisms used in GPUs to deal with the massive number of threads (e.g. *threadblock*-scheduling in CUDA [9]), increase the probability of deadlock to occur, namely active threads may be waiting for not-yet-scheduled threads due to blocking synchronization while the latter are waiting for the former to finish due to non-preemptive scheduling. Group-scheduling mechanisms used within an SIMD core (or *warp*-scheduling in CUDA terminology) also increase the probability of deadlock. Due to the SIMD architecture, different execution paths of a *divergent* code (e.g. one containing *if*-statement) must be serialized. If a barrier is used in different paths of the code executed by the same thread-group (or *warp* in CUDA terminology), deadlock will occur. The challenge is that since GPUs are optimized for processing 3D graphics (e.g. graphics rendering), a kind of data-parallel applications, several GPUs do not support

- *P. H. Ha and O. J. Anshus are with the Department of Computer Science, Faculty of Science, University of Tromsø, NO-9037 Tromsø, Norway. E-mail: {phuong, otto}@cs.uit.no*
- *P. Tsigas is with the Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden. E-mail: philippas.tsigas@chalmers.se*

strong synchronization primitives such as *compare-and-swap* (e.g. OpenCL architecture [2] and NVIDIA Tesla C870 and Quadro FX 5600 GPUs with 16 cores [3]), which are usually used to construct fault-tolerant/strong synchronization mechanisms.

This paper aims at bridging the gap between the lack of strong synchronization primitives in the GPUs and the need for strong synchronization mechanisms in parallel applications. Based on the intrinsic features of typical GPU architectures (e.g. OpenCL [2] and CUDA [3]), we first generalize the architectures to an abstract model of an MIMD[1] chip with multiple SIMD cores sharing a memory (cf. Section 2). Then, we construct wait-free and t-resilient synchronization objects [6], [10] for this model. The wait-free and t-resilient objects can be deployed as building blocks in parallel programming to help parallel applications tolerate crash-failures and gain performance.

We observe that due to SIMD architecture, each SIMD core with $\mathcal{M}$ hardware threads can read/write $\mathcal{M}$ memory locations in one *memory transaction*. For instance, in CUDA [3], simultaneous memory accesses to the global shared memory by threads of an SIMD core, during the execution of a read/write instruction, will be *coalesced* into a *single memory transaction* if coalescing conditions are satisfied (cf. Appendices G.3.2 and G.4.2 in [3]). For a comprehensive analysis of the coalesced memory access, the reader is referred to [11]. Note that this atomic access to $\mathcal{M}$ memory locations results from appropriately *coordinating* the accesses of $\mathcal{M}$ *threads* of an SIMD core, and thus this atomic access is clearly not a conventional synchronization primitive such as *test-and-set* and *compare-and-swap* that can be invoked by *each thread*. Compared with the conventional *m-register operation* in the literature [6], which allows *a thread* to write to *m arbitrary* registers atomically, this atomic access is M-register operation to *each SIMD core with $\mathcal{M}$ threads*, where $M \leq \mathcal{M}$ due to the conditions for coalescing to occur. Value M increases when the coalescing conditions are relaxed (cf. coalescing conditions for different versions of CUDA with respect to compute capability 1.1, 1.2 and 2.0 in Appendices G.3.2 and G.4.2 in [3]). For the sake of readability, we use the conventional term *M-register operation* in the literature to denote this atomic access for each SIMD core with $\mathcal{M}$ hardware threads, but note that the number M of registers in the atomic operation may be less than the number $\mathcal{M}$ of hardware threads of an SIMD core.

Although building synchronization objects using M-register read/write operations has been reported in the previous work [6], this paper improves the previous work [6] in several aspects. First, this paper shows how coalesced memory accesses on GPUs provide atomic M-register read/write operations. Second, unlike the *short-lived* consensus (SLC) object in [6] where the object variables are used *once* during the object lifetime, the *long-lived* consensus (LLC) object in this paper must allow processes to *re-use* the object variables so as to keep the object size bounded. This implies that the LLC object must include a wait-free/resilient memory management mechanism [12], [13] inside itself. Third, the new LLC object has the optimal space complexity $O(N^2)$ and access time complexity $O(N)$, which is better than the access time complexity $O(N^2)$ of the SLC object in [6] [2]. Finally, unlike the proposal used in the SLC object, the new wait-free *read-modify-write* (RMW) objects in this paper, which are built on the new LLC objects, must handle the proposal that is too large to be stored within one register. Since M-register assignment can atomically write M values to M memory locations only if each value can be stored in one register, the RMW objects must handle the proposal-size issue while tolerating the same number of crash failures $(2M-3)$ as the SLC object.

The main contribution of this paper is a new formal model for GPU computing and novel wait-free synchronization mechanisms for the GPU computing model, empowering the programmer with the necessary and sufficient tools for wait-free programming on graphics processors without synchronization primitives such as *test-and-set* and *compare-and-swap*. The technical contributions of this paper are threefold:

- We develop a wait-free *long-lived* consensus (LLC) object for $N = (2M-2)$ processes using only M-register read/write operations and read/write registers (cf. Section 3). The new LLC object guarantees *weaker* semantics than previous long-lived consensus protocols [14], namely the execution on the LLC object is organized as a (infinite) sequence of *round*s and the LLC object guarantees consensus for only processes participating in the *latest* round at the moment the LLC object is invoked (cf. Definition 2.4). The processes that do not participate in the latest round, are considered faulty. Surprisingly, the LLC object with such weak semantics is powerful enough to construct any wait-free read-modify-write (RMW) object for N processes (cf. Section 4). To the best of our knowledge, long-lived consensus objects with such weak semantics have not been reported previously. The new LLC algorithm has optimal space complexity $O(N^2)$ and time complexity $O(N)$, which are better than the time complexity $O(N^2)$ of the well-known *short-lived* consensus (SLC) algorithm [6].
- We develop a wait-free long-lived *read-modify-write* (RMW) object for $N = (2M-2)$ processes using only M-register read/write operations and read/write registers (cf. Section 4). The RMW object is basically built on top of the new LLC object. Accesses to the RMW object have time complexity $O(N)$. The RMW object has the optimal space complexity $O(N^2)$. This result implies that it is possible to construct wait-free synchronization mechanisms for GPUs without hardware synchronization primitives such as *test-and-set* and *compare-and-swap*.
- We develop a $(2M-3)$-resilient long-lived RMW (ResilientRMW) object for an arbitrary number N of pro-

1. MIMD: Multiple-Instruction-Multiple-Data

2. The SLC object needs to construct a directed graph of processes, leading to the time complexity $O(N^2)$

cesses using only M-register read/write operations and read/write registers (cf. Section 5). The $(2M-3)$-resilient RMW object is built on top of both the wait-free RMW object and the wait-free LLC object for $(2M-2)$ processes.

The rest of this paper is organized as follows. Section 2 presents a general model of an MIMD chip with multiple SIMD-cores on which the new wait-free/resilient objects are developed. Section 3 presents the wait-free long-lived consensus object for $N = (2M-2)$ processes. Section 4 presents the wait-free read-modify-write (RMW) object for $N = (2M-2)$ processes. Section 5 presents the $(2M-3)$-resilient RMW object for an arbitrary number N of processes. Finally, Section 6 concludes this paper. The new wait-free objects have been implemented and evaluated on commodity NVIDIA graphics cards. Due to the space constraint, the reader is referred to [15] for the detailed experimental study.

2 The Model

Inspired by emerging media/graphics processing unit architectures such as OpenCL [2], CUDA [3] and Cell BE [16], the abstract system model we consider in this paper is illustrated in Fig. 1. The model consists of N SIMD-cores $P_0, \ldots, P_{N-1}$ sharing R registers (or memory words) $V_0, \ldots, V_{R-1}$ and each core can process $\mathcal{M}$ threads $T_0, \ldots, T_{\mathcal{M}-1}$ (in an SIMD manner) in one clock cycle. For instance, NVIDIA GeForce 8800GTX graphics processor has 16 SIMD-cores/SIMD-multiprocessors, each of which processes up to 16 concurrent threads in one clock cycle of the SIMD core.

Using terminologies in the literature [17], we model SIMD cores as (nondeterministic) state machines and model executions as alternating sequences of configurations and events.

A *configuration* in the model is a vector

$$C = (p_0, \ldots, p_{N-1}, v_0, \ldots, v_{R-1})$$

where p_i is a state of core P_i and v_j is a value of register V_j. In *initial configuration*, all cores are in their initial states and all registers have their initial values.

An *event* $\phi = [i, b_0 \ldots b_{\mathcal{M}-1}]$ in the model is a computation step by core P_i, where bit b_k determines whether thread T_k of P_i participates in the computation step. At each computation step by P_i, the following happens atomically:

- P_i determines the set S of its threads T_k that participate in a specific operation (i.e. $b_k = 1$), based on P_i's current state. The size of set S is at least one.
- Each thread T_k of S chooses a shared register to access with the operation, based on P_i's current state.
- Each thread T_k performs the operation on its chosen register. If more than one threads of S write to the same register, the value of the register after this step is one of the values written (nondeterministic) [3].

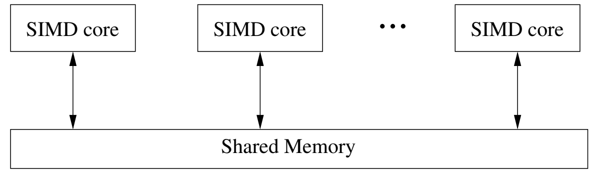

Fig. 1. The abstract model of an MIMD chip with multiple SIMD-cores

- P_i's state changes according to P_i's transition function, based on P_i's current state and the values returned by the operation to the threads of S.

An *execution segment* of an algorithm is an alternating sequence of configurations C_i and events ϕ_j:

$$E = C_0, \phi_1, C_1, \phi_2, C_2, \phi_3 \ldots$$

If $\phi_k = [i, b_0 \ldots b_{\mathcal{M}-1}]$ and P_i's state in C_{k-1} indicates that shared registers $V_j, \cdots, V_l, 0 \leq j < l \leq R-1$, to be accessed, C_k is the result of changing C_{k-1} according to P_i's computation step performing on P_i's state in C_{k-1} and the values of registers $V_j, \cdots, V_l$ in C_{k-1}. An *execution* is an execution segment that starts with an initial configuration.

In this model, SIMD cores access shared registers (or memory words) using only read-/write-operations. The shared memory is sequentially consistent (e.g. the global memory in CUDA GPUs with compute capability 1.0 supports sequential consistency for concurrent threads from different SIMD cores [18]). Since several graphics processors do not support strong synchronization primitives such as *test-and-set* and *compare-and-swap* (e.g. OpenCL specification [2] and CUDA GPUs with compute capability 1.0 [3]) , we make no assumption on the existence of such strong synchronization primitives in this model. In this model, each of the $\mathcal{M}$ threads of one SIMD core can read/write one register in one atomic step. Due to SIMD architecture, each SIMD core can read/write $\mathcal{M}$ different registers in one atomic step (e.g. coalesced memory accesses in CUDA [3]), namely each SIMD core can execute M_READ and M_ASSIGNMENT (atomic) operations [6], where the number M of registers in the atomic operation may be less than the number $\mathcal{M}$ of hardware threads of an SIMD core due to conditions for the memory transaction to occur (e.g. coalescing conditions in CUDA [3]). In CUDA [3], simultaneous memory accesses to words of an 128-byte memory segment by threads of an SIMD core, during the execution of a read/write instruction, will be coalesced into a *single memory transaction* (cf. Appendices G.3.2 and G.4.2 in [3]). That means in CUDA each SIMD core with $\mathcal{M}$ hardware threads, where $\mathcal{M} = 16$, can execute $\mathcal{M}$_READ and $\mathcal{M}$_ASSIGNMENT (atomic) operations on words of an 128-byte memory segment.

Different cores can concurrently execute different user programs. Let *process* be a sequential execution of computation steps of a program on one SIMD core. Namely, a process is comprised of $\mathcal{M}$ threads of an SIMD core and can execute M_READ and M_ASSIGNMENT operations, where $M \leq \mathcal{M}$. Processes are asynchronous and can crash due to the program errors. The failure category considered in this model is the crash failure: a failed process cannot take another computation step in the execution. This model supports the strongly t-resilient formulation in which the access procedure at some port[3] of an object is infinite only if the access procedures in more than t other ports of the object are finite, nonempty and incomplete in the object execution [19].

Definition 2.1 (Wait-freedom [6], [20], [21]): An object implementation is *wait-free* if any non-faulty process completes any operation on the object in a finite number of steps regardless of the execution speeds of other processes.

Definition 2.2 (t-resilience [22], [23]): An object implementation is *t-resilient* if non-faulty processes will complete their operations as long as no more than t processes fail, where t is a specified parameter.

Definition 2.3 (Short-lived consensus object): A short-lived consensus object allows each process p_i to propose an input from some set S, $|S| \geq 2$, and then returns an output to p_i so that the following properties are satisfied in *every* execution:

- *Wait-freedom*: each non-faulty process gets an output after a finite number of steps.
- *Agreement*: the outputs of all non-faulty processes are identical;
- *Validity*: the output of each non-faulty process is the input of some process;

In the *short-lived* consensus setting, processes start with their inputs and have to solve consensus *once*. In order to construct real data objects on which each process can execute an arbitrary sequence of operations, we need a *long-lived* consensus setting in which processes change inputs over time and have to solve consensus repeatedly. A *round* is intuitively the interval between two input changes. The definition of when a round starts and finishes, depends on specific algorithms that use the long-lived consensus setting. The long-lived consensus (LLC) object considered in this paper is required to satisfy the three aforementioned properties only for processes participating in the *latest* round at the moment the LLC object is invoked. The processes that do not participate in the latest round, are considered faulty in the latest round. The long-lived consensus object will be used to construct wait-free read-modify-write (RMW) objects in Section 4. The precise definition of the long-lived consensus object is as follows.

Definition 2.4 (Long-lived consensus object): Each process is associated with the latest round in which it participates. In each round, a long-lived consensus object allows each process p_i to propose an input from some set S, $|S| \geq 2$, and then returns an output to p_i so that the following properties are satisfied in *every* execution:

- *Wait-freedom*: each non-faulty process (regardless of the latest round participation) gets an output after a finite number of steps.
- *Agreement*: the outputs of all non-faulty processes participating in the latest round are identical;
- *Validity*: the output of each non-faulty process participating in the latest round is the input of some process participating in the same round;

Definition 2.5 (Read-modify-write object [24]): A read-modify-write object allows each process to read the object value X, update the object value to Y and return the old value X *atomically*.

Definition 2.6 (Consensus number [6]): The *consensus number* of an object type is either the maximum number of processes for which wait-free (short-lived) consensus can be solved using only objects of this type and registers [4], or infinity if such a maximum does not exist.

3 Wait-free Long-lived Consensus Objects Using $M_$assignment for $N = 2M - 2$

In this section, we consider the following consensus problem. Each process is associated with a round number before participating in a consensus protocol. The round number must satisfy Requirement 1 below. The problem is to construct a long-lived object that guarantees consensus among processes with the latest round number (or processes within the latest round) using $M_$ASSIGNMENT operation. Since i) the adversary can arrange all N processes to be in the latest round and ii) the $M_$ASSIGNMENT operation has consensus number $(2M-2)$ [6], we cannot construct any wait-free consensus objects that guarantee consensus for more than $(2M-2)$ processes using only the operation and read/write registers [6], or $N \leq (2M-2)$ must hold. The constructed wait-free long-lived consensus object will be used as a building block to construct wait-free read-modify-write objects in Section 4.

Requirement 1: The requirements for processes' round number:

- a process' round number must be increasing and be updated only by this process,
- processes get a round number r only if the round $(r-1)$ has finished [5] and
- processes declare their current round number in shared variables before participating in a consensus protocol.

For the sake of simplicity, round numbers are assumed to be unbounded. General solutions to bounding round numbers have been reported in [25], [26].

3.1 General descriptions

We now present a high-level description of the wait-free long-lived consensus (LLC) object for $N = (2M-2)$ processes using $M_$ASSIGNMENT operations. The detailed algorithms and correctness proofs are presented in Section 3.2.

3. An object that allows N processes to access concurrently is considered having N ports.

4. A *register* supports only read and write operations.

5. The definition of when a round finishes, depends on specific algorithms that use this long-lived consensus object.

The LLC object is developed from the short-lived consensus (SLC) object using M_ASSIGNMENT in [6]. The LLC object will be used to achieve an agreement among processes in the latest round. Unlike the SLC object, variables in the LLC object that are used in the current round can be reused in the next rounds. The LLC object, moreover, must handle the case that some processes (e.g. slow processes) belonging to other rounds try to modify the shared data/variables that are being used in the current round.

The algorithm of the wait-free LLC object using M_ASSIGNMENT is presented in Algorithm 1. Before a process p_i invokes the LONGLIVEDCONSENSUS procedure, p_i's round number must be declared in the shared variable r_i. The procedure returns i) $\perp$ if p_i's round had finished and a newer round started or ii) one of the proposal data proposed in p_i's round.

The LLC algorithm divides the group of $(2M-2)$ processes into two fixed equal subgroups of $(M-1)$ processes (line 1L). In the first phase, the invoking process p_i finds the proposal of the earliest process of its group in its current round (line 2L). Then in the second phase, p_i uses the agreement achieved among its group in the first phase as its proposal for finding an agreement with its opposite group in its round (line 6L). The data structures used in the two phases are one array of 2-writer registers $2WR[][]$ where element $2WR[i][j]$ can only be written by processes p_i and p_j, and two arrays of 1-writer registers $1WR[][0]$ and $1WR[][1]$ where $1WR[][0]$ is used in the first phase, $1WR[][1]$ is used in the second phase and elements $1WR[i][0], 1WR[i][1]$ can only be written by process p_i.

Figure 2 illustrates the LLC algorithm for 4 processes within the latest round using 3_assignment (i.e. $M=3$ and $N=4$). The algorithm divides the 4 processes into two groups $G_0=\{p_0,p_1\}$ and $G_1=\{p_2,p_3\}$. Consider group $\{p_0,p_1\}$. In the first phase, group $\{p_0,p_1\}$ uses one 2-writer register $2WR[1][0]$ and two 1-writer registers $1WR[0][0]$ and $1WR[1][0]$ to achieve an agreement within the group. Process p_0 proposes its index 0 by writing 0 atomically to two registers $2WR[1][0]$ and $1WR[0][0]$ using 3_assignment. Similarly, process p_1 proposes its index 1 by writing 1 atomically to two registers $2WR[1][0]$ and $1WR[1][0]$. Based on the values of the three registers written in the latest round, p_0 and p_1 determine who is the first process executing the 3_assignment and then agree on the proposal of the first process. The fact that the final value of $2WR[1][0]$ is 1, p_1's proposal (cf. Figure 2(a)), and p_0 has written 0 to $2WR[1][0]$ (since $1WR[0][0]=0$), indicates that p_1 has come after p_0 and overwritten $2WR[1][0]$ with p_1's proposal. Therefore, p_0 and p_1 agree on p_0's proposal 0 in the first phase and propose 0 in the second phase to find an agreement with the other group $\{p_2,p_3\}$. The details of the first phase are presented in Algorithm 2.

In the second phase (cf. Figure 2(b)), the four processes use four 2-writer registers $2WR[2][0]$, $2WR[3][0]$, $2WR[2][1]$, $2WR[3][1]$ and four 1-writer registers $1WR[0][1]$, $1WR[1][1]$, $1WR[2][1]$, $1WR[3][1]$, in order to agree on a proposal proposed by one of the two

Algorithm 1 LONGLIVEDCONSENSUS(buf_i: proposal) invoked by process p_i with round r_i

$REG[\,][\,]$ of $Integer$: 2-writer registers. $REG[i][j]$ can be written by processes p_i and p_j. Initially, $REG[i][j] \leftarrow \perp$. For the sake of simplicity, we use a virtual array $2WR[1\dots N][1\dots N]$ that has no elements $2WR[i][i]$ and is mapped to a strictly lower triangular matrix REG of size $\frac{N(N-1)}{2}$ as follows

$$2WR[i][j] = \begin{cases} REG[i][j] & \text{if } i > j \\ REG[j][i] & \text{if } i < j \end{cases}$$

$Privacy$: **record** $value, round$ **end**.
$1WR[1\dots N][0\dots 1]$ of $Privacy$: 1-writer registers. $1WR[i]$ can be written by process p_i only. Initially, $1WR[i][0] \leftarrow 1WR[i][1] \leftarrow \langle \perp, \perp \rangle$.

Input: p_i's unique proposal buf_i **and** p_i's round number r_i.
Output: a proposal **or** $\perp$.
1L: $gId \leftarrow \lfloor \frac{i}{M} \rfloor$ // Divide processes into 2 groups of size $(M-1)$ with group ID $gId \in \{0,1\}$
// **Phase I:**Find an agreement in p_i's group with indices $\{gId(M-1)+1, \cdots, gId(M-1)+M-1\}$
2L: $first \leftarrow$ FIRSTAGREEMENT(buf_i, gId) // $first$ is the proposal of the earliest process of group gId in p_i's round
3L: **if** $first = \perp$ **then**
4L: **return** $\perp$ // p_i's round had finished and a new round has started
5L: **end if**
// **Phase II**: Find an agreement with the other group with indices $\{(\neg gId)(M-1)+1, \cdots, (\neg gId)(M-1)+M-1\}$
6L: $winner \leftarrow$ SECONDAGREEMENT$(first, gId)$
7L: **if** $winner = \perp$ **then**
8L: **return** $\perp$ // p_i's round had finished and a new round has started
9L: **end if**
10L: **return** $winner$

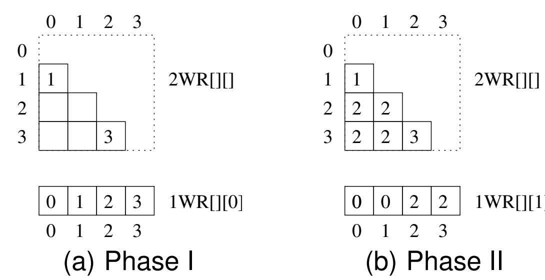

Fig. 2. Illustration for the LLC algorithm with 4 processes.

groups. Assume that group $\{p_2,p_3\}$ proposes 2. Process p_0 proposes its group's proposal 0 by writing 0 atomically to three registers $2WR[2][0], 2WR[3][0]$ and $1WR[0][1]$ using 3_assignment. Similarly, process p_1 writes 0 atomically to three registers $2WR[2][1], 2WR[3][1]$ and $1WR[1][1]$. Process p_2 proposes its group's proposal 2 by writing 2 atomically to three registers $2WR[2][0], 2WR[2][1]$ and $1WR[2][1]$. Similarly, process p_3 writes 2 atomically to three registers $2WR[3][0], 2WR[3][1]$ and $1WR[3][1]$. Consider process p_0 of group G_0 and processes p_2, p_3 of group G_1. Based on the values of registers $2WR[2][0], 2WR[3][0]$ written by p_0, p_2 and p_3 in the last round, processes p_0, p_2 and p_3 can determine which group is the first group executing the 3_assignment in the second phase and then agree on the proposal of the first group. In Figure 2(b), the final value of $2WR[2][0]$ is 2, p_2's proposal, indicating that p_2 has come after p_0 and overwritten $2WR[2][0]$ with p_2's proposal. Similarly, $2WR[3][0]=2$ indicates that p_3 has

Algorithm 2 FIRSTAGREEMENT(buf_i: proposal; gId: bit) invoked by process p_i with round r_i

Output: $\perp$ **or** the proposal of the earliest process in p_i's round
1F: M_ASSIGNMENT($\{1WR[i][gId], 2WR[i][\alpha+1], \cdots, 2WR[i][\alpha+M-1]\}, \{\langle buf_i, r_i\rangle, buf_i, \cdots, buf_i\}$), where $\alpha = gId(M-1)$ // $2WR[i][i]$ is not written.
2F: $first \leftarrow i$ // Initialize the winner $first$ of p_i's group to p_i
3F: **for** k in $\alpha+1, \cdots, \alpha+M-1$ **do**
4F: $\quad \{first, ref\} \leftarrow$ ORDERING($first, k, gId$) // Find the earliest process $first$ of p_i's group in p_i's round
5F: $\quad$ **if** $first = \perp$ **then**
6F: $\quad\quad$ **return** $\perp$ // p_i's round had finished and a new round has started
7F: $\quad$ **end if**
8F: **end for**
9F: **return** ref // $first$'s proposal in p_i's round

come after p_0. That means G_0 is the first group that executes the 3_assignment in the second phase and thus p_0, p_2 and p_3 agree on G_0's proposal 0. Similarly, p_1, p_2 and p_3 also agree on G_0's proposal 0 by looking at $2WR[2][1], 2WR[3][1]$. The details of the second phase are presented in Algorithm 3.

3.2 Detailed algorithms and correctness proofs

Definition 3.1 (Single-group order $\rightsquigarrow_s$): Suppose two processes p_i and p_j belong to the same subgroup G and are in the same round r. Process p_i *precedes* p_j (denote $p_i \rightsquigarrow_s p_j$) in round r iff p_i executes its M_ASSIGNMENT on their shared register $2WR[i][j]$ (line 1F in Algorithm 2) before p_j in round r.

Definition 3.2 (Different-group order $\rightsquigarrow_d$): Suppose two processes p_i and p_j belong to different subgroups G and $\neg G$, and are in the same round r. Process p_i *precedes* p_j (denote $p_i \rightsquigarrow_d p_j$) in round r iff p_i executes its M_ASSIGNMENT on their shared register $2WR[i][j]$ (line 1S in Algorithm 3) before p_j in round r.

Note that the single-group order and different-group order do not define an order over *all* processes, but only an order over either two processes of the same subgroup (i.e. single-group order) or two processes of different subgroups (i.e. different-group order).

Definition 3.3 (Earliest process): The *earliest* process of a subgroup G in round r is the process that precedes the rest of G in round r according to the single-group order.

Note that p_i's round number is unchanged when p_i is executing the LONGLIVEDCONSENSUS procedure. If p_i's round has already finished, the procedure returns $\perp$ since p_i is not allowed to participate in a consensus protocol of a round to which it doesn't belong (lines 4L and 8L).

The FIRSTAGREEMENT procedure (cf. Algorithm 2), after executing M_ASSIGNMENT (line 1F), simply scans all members of p_i's group to find the earliest process in p_i's round using the ORDERING procedure (cf. Algorithm 4). The ORDERING procedure receives as input two processes $first$ and k, and returns the preceding one together with its proposal in p_i's round (cf. Lemma 3.4). If both processes $first$ and k belong to p_i's round, the preceding process is the one that first executes its M_ASSIGNMENT (line 1F). If process k belongs to a previous round, it is

Algorithm 3 SECONDAGREEMENT($first$: proposal; gId: bit) invoked by process p_i with round r_i

1S: M_ASSIGNMENT($\{1WR[i][\neg gId], 2WR[i][\beta+1], \cdots, 2WR[i][\beta+M-1]\}, \{\langle first, r_i\rangle, first, \cdots, first\}$), where $\beta = (\neg gId)(M-1)$ // $2WR[i][i]$ is not written.
2S: $winner \leftarrow i$ // Initialize the winner $winner$ to p_i
3S: $w\_gId \leftarrow gId$ // Initialize the winner's group ID w_gId
4S: $pivot[w\_gId] \leftarrow i$ // Set pivots for both groups to check all members of each group in a round-robin manner
5S: $pivot[\neg w\_gId] \leftarrow \beta+1$ // The smallest index in $winner$'s opposite group
6S: $next \leftarrow pivot[\neg w\_gId]$
7S: **repeat**
8S: $\quad previous \leftarrow winner$
9S: $\quad \{winner, ref\} \leftarrow$ ORDERING($winner, next, \neg w\_gId$)
10S: $\quad$ **if** $winner = \perp$ **then**
11S: $\quad\quad$ **return** $\perp$ // p_i's round had finished and a new round has started
12S: $\quad$ **else if** $winner \neq previous$ **then**
13S: $\quad\quad w\_gId \leftarrow \neg w\_gId$ // $winner$ now belongs to the other group
14S: $\quad\quad next \leftarrow previous$
15S: $\quad$ **end if**
16S: $\quad next \leftarrow$ the next member index in $next$'s group in a round-robin manner.
17S: **until** $next = pivot[\neg w\_gId]$ // All members of $winner$'s opposite group have been checked
18S: **return** ref // $winner$'s proposal in round $round_i$

Algorithm 4 ORDERING($first, k$: index; gId: bit) invoked by process p_i with round r_i

Output: $\{\perp,\perp\}$ or {index, proposal}
1O: $1wr_k \leftarrow 1WR[k][gId]$; $2wr_{first,k} \leftarrow 2WR[first][k]$; $1wr_{first} \leftarrow 1WR[first][gId]$ // Registers are read sequentially from left to right.
2O: **if** $(r_i < 1wr_{first}.round)$ **or** $(r_i < 1wr_k.round)$ **then**
3O: $\quad$ **return** $\{\perp,\perp\}$ // A newer round has started $\Rightarrow$ p_i's round had finished.
4O: **else if** $1wr_{first}.round > 1wr_k.round$ **then**
5O: $\quad$ **return** $\{first, 1wr_{first}.value\}$ // $r_i = 1wr_{first}.round$ and $1wr_k.round$ has finished $\Rightarrow$ Ignore $1wr_k$.
6O: **end if**
// $r_i = 1wr_{first}.round = 1wr_k.round$.
7O: **if** $2wr_{first,k} = 1wr_k.value$ **then**
8O: $\quad$ **return** $\{first, 1wr_{first}.value\}$
9O: **else**
10O: $\quad$ **return** $\{k, 1wr_k.value\}$
11O: **end if**

considered a faulty process in p_i's round and is ignored by the ORDERING procedure. Since the preceding order is transitive, the variable $first$ after the for-loop is the earliest process of p_i's group in p_i's round. Since FIRSTAGREEMENT scans $(M-1)$ processes of p_i's group to find the earliest one and ORDERING has time complexity $O(1)$, FIRSTAGREEMENT has time complexity $O(M)$ (or $O(N)$).

The SECONDAGREEMENT procedure (cf. Algorithm 3) is an innovative improvement of the abstract idea in the SLC algorithm [6]. The SLC algorithm suggests the idea of constructing a directed graph between two groups each of $(M-1)$ processes with property that there is an edge from P_l to P_k if P_l and P_k are in different groups and P_l's assignment precedes P_k's (or P_l precedes P_k for short). Constructing such a directed graph has time complexity $O(M^2)$ since each member of one group must be checked with $(M-1)$ members of the other group.

However, the SECONDAGREEMENT procedure finds an agreement with time complexity only $O(M)$. The idea is

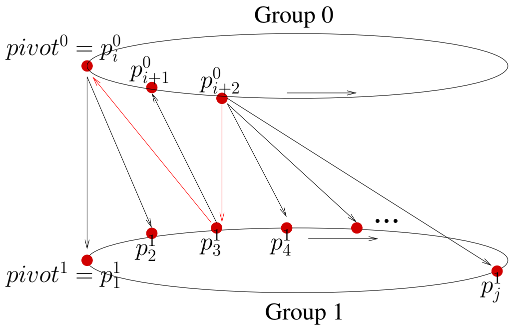

Fig. 3. Illustration for the SECONDAGREEMENT procedure, Algorithm 3

that we can find a process p_w in a group G_0 that precedes all members of the other group G_1 without the need for such a directed graph. Such a process is called *source*. Since all members of G_1 are preceded by p_w, they cannot be sources. All sources must be members of p_w's group G_0, which suggest the same proposal, their agreement achieved in the first phase. Therefore, all processes in both groups will achieve an agreement, the agreement of p_w's group.

The SECONDAGREEMENT procedure utilizes the transitive property of the preceding order to achieve the better time complexity $O(M)$. Fig. 3 illustrates the procedure. Assume that process p_i belongs to group 0, which is marked as p_i^0 in the figure. The procedure sets a pivot index for each group (e.g. $pivot^0 = p_i^0$ and $pivot^1 = p_1^1$) and checks members of each group in a round-robin manner starting from the group's pivot (lines 4S and 5S). In the figure, p_i^0, which is the temporary winner (line 2S), consecutively checks the members of group 1: p_1^1, p_2^1 and p_3^1, and discovers that it precedes p_1^1 and p_2^1 but it is preceded by p_3^1 according to the different-group order (cf. Definition 3.2). At this point, the temporary winner $winner$ is changed from p_i^0 to p_3^1 and p_3^1 starts to checks the members of group 0 starting from p_{i+1}^0 (lines 12S-14S). Then, p_3^1 discovers that it precedes p_{i+1}^0 but it is preceded by p_{i+2}^0. At this point, the temporary winner $winner$ is again changed from p_3^1 to p_{i+2}^0. p_{i+2}^0 continues to check the members of group 1 *starting from* p_4^1*, the index before which* p_i^0 *stopped, instead of starting from* $pivot^1 = p_1^1$ (lines 12S-14S). It is clear from the figure that p_{i+2}^0 precedes p_1^1 and p_2^1 (or $p_{i+2}^0 \rightsquigarrow_d p_1^1$ and $p_{i+2}^0 \rightsquigarrow_d p_2^1$ for short) since $p_{i+2}^0 \rightsquigarrow_d p_3^1 \rightsquigarrow_d p_i^0$ and p_i^0 precedes both p_1^1 and p_2^1. Therefore, as long as the temporary winner (e.g. p_{i+2}^0) checks the $pivot$ of its opposite group again, it can ensure that it precedes all the members of its opposite group (line 17S) and becomes the final winner. Therefore, the procedure needs to check at most $(2M-2)$ times, leading to the time complexity $O(M)$. This argument also leads to the following lemma.

Lemma 3.1: The process $winner \neq \perp$ whose ref is returned by SECONDAGREEMENT precedes all processes of the other group.

Proof: Let the final $winner$ ($\neq\perp$) be $\mathcal{W}$. Since the ORDERING procedure returns the preceding process of two processes $winner$ and $next$ in the same round $round_i$ (cf. Lemma 3.4), the final winner $\mathcal{W}$ precedes all processes in $round_i$ that are checked in the repeat-until loop (lines 7S - 17S) according to the different-group order (cf. Definition 3.2). What we need to prove is that all processes in the other group $\neg w\_gId$ have been checked in the loop. Indeed,

- if the $winner$ has never been changed (i.e. $winner = previous$ all the time), the next member of the group $\neg w\_gId$ in a round-robin manner (line 16S) will be checked against $winner$ until the repeat-until loop makes a complete check on all members of the group $\neg w\_gId$ (line 17S).
- if the $winner$ has ever changed to a member $\tilde{W}$ of the other group $\neg w\_gId$, $\tilde{W}$ will continue to check the next member after the previous winner $previous$ in a round-robin manner (lines 14S and 16S) until either all members of $\tilde{W}$'s opposite group have been checked within the loop or a member of $\tilde{W}$'s opposite group precedes $\tilde{W}$. That means in each iteration, regardless of whether $winner$ is changed or not, the next member in one of the two groups will be checked in a round-robin manner, starting from the group pivot (lines 4S and 5S). Since members of each group is checked consecutively and the loop finishes when the pivot of a group $\tilde{G}$ is checked again, all members of the group $\tilde{G}$ are checked when the loop finishes. The fact that the final winner $\mathcal{W}$ belongs to the other group $\mathcal{G} \neq \tilde{G}$ when the loop finishes, implies that all members of $\mathcal{W}$'s opposite group have been checked in the loop.

□

Lemma 3.2: The SECONDAGREEMENT procedure has time complexity $O(N)$.

Proof: As shown in the proof of Lemma 3.1, in each iteration of the repeat-until loop (lines 7S-17S), regardless of whether $winner$ is changed or not, the next member in one of the two groups will be checked in a round-robin manner, starting from the group pivot (lines 4S and 5S). Therefore, the repeat-until loop has at most N iterations. Since the time complexity of ORDERING is $O(1)$, the time complexity of SECONDAGREEMENT is $O(N)$. □

We now show that the values of shared variables (e.g. $2WR$ and $1WR$) used by the ORDERING procedure (Algorithm 4) are written by processes in p_i's round. Such values are considered *belonging* to p_i's round. The procedure ensures that by checking the $round$ field of the $1WR$ variables.

Lemma 3.3: Let α be any execution which contains the execution of some instance ord of the ORDERING procedure (Algorithm 4) by some process p_i with round r_i. Then,

1) at all configurations of α following the execution of line 4O by ord and preceding the response of ord, it holds that $1wr_{first}.round = r_i$ and $1wr_{first}.round \geq 1wr_k.round$, and
2) at all configurations following the execution of line 7O by ord and preceding the response of ord, it holds that $1wr_k.round = 1wr_{first}.round = r_i$

Proof: We first prove that in all configurations of α during the execution of ord, it holds that $1wr_{first}.round \geq r_i$. Let $first_0, first_1, first_2, \ldots$ be instances of parameter $first$ of ORDERING $(first, k, gId)$ invoked by p_i during the loop 3F-8F in Algorithm 2 or the loop 7S-17S in Algorithm 3, where $first_0 = i$ (line 2F or 2S). We will prove by induction on index $j \geq 0$ that $1wr_{first_j}.round \geq r_i$.

- The hypothesis holds when $j = 0$. Indeed, we have $first_0 = i$. Since i) p_i writes r_i to $1WR[i][gId].round$ (line 1F or 1S) before calling ORDERING$(first, k, gId)$ (line 4F or 9S) and ii) p_i's round r_i is unchanged while p_i is executing LONGLIVEDCONSENSUS (cf. Requirement 1, items 1 and 3), it holds that $1wr_{first_0}.round (= 1wr_i.round) \geq r_i$.
- We now prove that if $1wr_{first_j}.round \geq r_i$ then $1wr_{first_{j+1}}.round \geq r_i$. Indeed, $first_{j+1} \neq \perp$ is the index returned from ORDERING$(first_j, k, gId)$ invoked by p_i with round r_i (line 4F or 9S). Index $first_{j+1}$ is either $first_j$ (line 5O or 8O) or k (line 10O). If $first_{j+1} = k$, $1wr_k.round \geq 1wr_{first_j}.round$ holds (otherwise, ORDERING$(first_j, k, gId)$ returned earlier at line 5O). In all cases, $1wr_{first_{j+1}}.round (\geq 1wr_{first_j}.round) \geq r_i$. Note that $p_{first_{j+1}}$'s round number only increases (cf. Requirement 1, item 1) (and thus $1WR[first_{j+1}][gId].round$ only increases) and p_i's round r_i is unchanged while p_i is executing LONGLIVEDCONSENSUS.

Therefore, from line 4O, $1wr_{first}.round = r_i$ and thus $1wr_k.round \leq (r_i =)\ 1wr_{first}.round$ (otherwise, the procedure returned early at line 3O). It follows that from line 7O, $1wr_k.round = 1wr_{first}.round = r_i$ (otherwise, the procedure returned early at line 5O). □

Lemma 3.4: The ORDERING procedure returns

- $\{\perp, \perp\}$ iff a newer round than p_i's round r_i has started (which implies that r_i had finished), or
- a pair $\{index, proposal\}$ in which $proposal$ is p_{index}'s proposal in round r_i and $index$ is either i) the preceding process between p_{first} and p_k in the case that both $1wr_{first}$ and $1wr_k$ belong to p_i's round r_i, or ii) $first$ in the case that $1wr_k.round$ has finished and $1wr_{first}.round = r_i$.

Proof: The first part of this lemma is clear from the ORDERING pseudocode. The procedure returns $\{\perp, \perp\}$ iff a newer round than r_i has started (line 3O).

We now prove that $proposal$ returned belongs to p_i's round r_i. The procedure returns a pair $\{index, proposal\}$ only at lines 5O, 8O and 10O, where $proposal$ is p_{index}'s proposal. Since $1wr_{first}.round = r_i$ from line 4O (cf. Lemma 3.3), $1wr_{first}.value$ returned at lines 5O and 8O belongs to r_i. Since $1wr_k.round = r_i$ from line 7O (cf. Lemma 3.3), $1wr_k.value$ returned at line 10O belongs to r_i.

We prove the last part of the lemma. It is clear from the ORDERING pseudocodes that the ORDERING procedure returns $first$ at line 5O only if $1wr_k.round$ has finished and $1wr_{first}.round = r_i$. In the case $1wr_{first}.round = 1wr_k.round = r_i$ (i.e from line 7O), both processes p_{first} and p_k have executed their corresponding M_ASSIGNMENT (line 1F in Algorithm 2 or 1S in Algorithm 3) in round r_i, which means $2wr_{first,k}$ is either $1wr_{first}.value$ or $1wr_k.value$. Note that equation $1wr_k.round = r_i$ implies that process p_k has executed its corresponding M_ASSIGNMENT with round r_i before process p_i reads $1WR[k][gId]$ at line 1O (Algorithm 4). According to the proof of Lemma 3.3, process p_{first} has written its round $r_{first} \geq r_i$ to $1WR[first][gId]$ (using M_ASSIGNMENT) before p_i invokes ORDERING$(first, k, gId)$. That means, in the case $1wr_{first}.round = 1wr_k.round = r_i$, both processes p_k and p_{first} have executed their corresponding M_ASSIGNMENT with round r_i *before* process p_i reads $1WR[k][gId]$ (as well as $2WR[first][k]$ and $1WR[first][gId]$) at line 1O (Algorithm 4). Therefore, if $2wr_{first,k} = 1wr_k.value$, process k has come after the process $first$ and has overwritten $2wr_{first,k}$. As a result, process $first$ is the preceding and is returned (line 8O). Otherwise, k is the preceding and is returned (line 10O). Note that the proposal data are unique for each process. □

Lemma 3.5: The time complexity of the LONGLIVEDCONSENSUS procedure is $O(N)$.

Proof: The time complexity of ORDERING is $O(1)$. Since FIRSTAGREEMENT scans $(M-1)$ processes of p_i's group to find the earliest one, its time complexity is $O(M)$ (or $O(N)$). Since SECONDAGREEMENT checks at most N processes in the repeat-until loop (cf. Fig. 3), its time complexity is also $O(N)$ (cf. Lemma 3.2). Therefore, the time complexity of LONGLIVEDCONSENSUS is $O(N)$. □

Lemma 3.6: The new object (cf. Algorithm 1) is long-lived and solves wait-free consensus for processes within the latest round in a system of $N = 2M - 2$ processes.

Proof: Since the shared data structures in the object are reused during the object lifetime, the new object is long-lived. We now prove that the new object satisfies consensus properties *agreement*, *validity* and *wait-freedom* for processes within the latest round.

- *Agreement*: Since processes $winner \neq \perp$ whose proposals are returned from SECONDAGREEMENT invoked by processes p_i with the latest round r, precede all processes of the other group (cf. Lemma 3.1), the processes $winner$ in round r must belong to the same group gId. Therefore, their proposals $first \neq \perp$ for SECONDAGREEMENT in round r are the same, which are the result of their FIRSTAGREEMENT (line 2L). Note that FIRSTAGREEMENT and SECONDAGREEMENT invoked by processes p_i with the *latest* round r never return $\perp$ (cf. Lemma 3.4). That means the values returned by LONGLIVEDCONSENSUS to all processes within the latest round (line 10L) are the same.
- *Validity*: Since ORDERING$(first, k, gId)$ invoked by a process p_i with the *latest* round r returns the proposal of either $first$ or k (Lemma 3.4), the value $first$ returned from FIRSTAGREEMENT invoked by p_i (line 2L in Algorithm 1) is an original proposal of some process of p_i's group in round r (cf. the for-loop 3F-8F in Algorithm 2). Similarly, the value $winner$ returned

from SECONDAGREEMENT invoked by p_i (line 6L in Algorithm 1) is either $first$ or an original proposal of some process of p_i's opposite group in round r (cf. the repeat-until loop 7S-17S in Algorithm 3). Therefore, the value $winner$ returned from LONGLIVEDCONSENSUS invoked by p_i with the *latest* round r is an original proposal of some process in round r.

- *Wait-freedom*: Since LONGLIVEDCONSENSUS has time complexity $O(N)$ (Lemma 3.5), each process invoking LONGLIVEDCONSENSUS will get a value after a finite number of steps.

□

Lemma 3.7: For any wait-free consensus protocols using only the $M_$ASSIGNMENT operation and read/write registers, the space complexity is $\Omega(N^2)$.

Proof: It has been proven that in any wait-free consensus protocols using only the $M_$ASSIGNMENT operation and read/write registers, each pair of processes having different proposal values must have a register that is written only by those two processes (cf. the proof of Theorem 13 in [6]). Therefore, for N processes there must be at least $\frac{N(N-1)}{2}$ registers, which means that the space complexity is $\Omega(N^2)$. □

Lemma 3.8: The space complexity of the LLC object is $O(N^2)$, the optimal.

Proof: From the set of variables used to construct the LLC object (cf. Algorithm 1), the space complexity of the LLC object is obviously $O(N^2)$ due to array REG. Due to Lemma 3.7, the space complexity of the LLC object is optimal. □

4 WAIT-FREE READ-MODIFY-WRITE OBJECTS FOR $N = 2M - 2$

In this section, we present a wait-free read-modify-write (RMW) object for $N = (2M - 2)$ processes using M_ASSIGNMENT operations. Since the M_ASSIGNMENT operation has consensus number $(2M - 2)$, we cannot construct any wait-free objects for more than $(2M - 2)$ processes using only this operation and read/write registers [6].

The idea is to divide the execution of the RMW object by processes p_i into consecutive rounds based on p_i's rounds. Each process p_i is associated with a round number r before trying to execute a function f on the RMW object. If p_i fails to execute its function f in round r, p_i will retry to execute its function again with a new round number $r' > r$ (cf. Lemma 4.6). Processes with the same round number (or in the same round) each suggests an order of these processes' functions to be executed on the object in that round, and then invokes the LONGLIVEDCONSENSUS procedure in Section 3 to achieve an agreement among these processes. Since each process executes one function on the RMW object at a time, functions are ordered according to both the round in which their matching processes participate, and the agreed order among processes in the same round.

Definition 4.1: A function is considered *executed* in a round iff its result is made within that round.

Algorithm 5 Data structures and variables used in Algorithm 6

$Proposal$: **record** $owner, round, response[1..N], toggle[1..N], value$ **end**. Initially, $toggle[] \leftarrow \{0\}$.
$Proposal_{ref}$: reference to $Proposal$;
$Buffer$: **record** $curBuf, PRO[0..1]$ of $Proposal$ **end**; Initially, $curBuf \leftarrow 0$.
$BUF[1...N]$ of $Buffer$: In $BUF[i]$, $PRO[curBuf]$ is the current buffer for p_i's proposed data, which is called $PRO_i[curBuf]$ for short. $PRO_i[\neg curBuf]$ is p_i's currently shared (read-only) buffer. Only p_i can write to $BUF[i]$
$WINNER[1...N]$ of $Proposal_{ref}$: $WINNER[i]$ contains the reference/address of the buffer containing the agreed proposal in the latest round in which p_i participates. Only p_i can write to $WINNER[i]$. Initially, $WINNER[i] \leftarrow \perp$.
$Function$: **record** $func, toggle$ **end**. Initially, $toggle \leftarrow 0$.
$FUN[1...N]$ of $Function$: $FUN[i]$ contains the function most recently suggested by process p_i. Only p_i can write to $FUN[i]$.
$COU[1...N]$: $COU[i]$ contains the latest round p_i has finished. Only p_i can write to $COU[i]$. Initially, $COU[] \leftarrow 0$.
FASTSCAN(): scans a set of size less than $2M$ using the M-register read/write operations. Its time complexity and space complexity are $\Theta(1)$ [27]

Definition 4.2: A process is considered *participating* in a round iff its function is executed in that round.

Definition 4.3: A function f is *executed by a process* p in a round r iff f is included in p's proposal and p is the winner of the long-lived consensus protocol among the participating processes of the round r.

Particularly, a process p_i, which wants to execute a function f on the RMW object, invokes the RMW procedure (Algorithm 6) with function f as its parameter. The function, together with a toggle bit, is written to a shared variable $FUN[i]$ so as to inform other processes (line 2). $FUN[i]$ is read-only for other processes $p_j, j \neq i$. Processes, when making a proposal, will scan all N elements of FUN to extract the functions that have not been executed yet based on their toggle bit (lines 19 and 21) and apply the functions on their local copies of the RMW object in the order imposed by process ids (line 23). Since each process executes one function on the RMW object at a time, the toggle bit is sufficient to check if a process' current function has been executed (cf. Lemma 4.7). A local copy LC_i of the RMW object by some process p_i will become the actual RMW object when processes agree on p_i's proposal using LONGLIVEDCONSENSUS from Section 3. Processes p_j then keep the reference to LC_i (or p_i's proposal) in $WINNER[j]$ (line 38). The functions that are applied at each round are those read in FUN by the winner p_i of the round.

In order to use the LONGLIVEDCONSENSUS procedure, each process needs to manage its own round number, which is increasing. For the sake of simplicity, round numbers are assumed to be unbounded [6]. A process p_i records the latest round it has finished in variable $COU[i]$, which is read-only to other processes $p_j, j \neq i$ (line 38).

Definition 4.4: A round r *starts* with the first process that obtains round number r (line 4). A round r is considered *finished* as soon as r is recorded in a variable

6. General solutions to bounding round numbers have been reported in [25], [26].

$COU[i]$ by a process p_i (line 38).

The process p_i, when invoking RMW, first scans all N elements of COU to find the most recent round number $round_i$, the round it will belong to (line 4). This ensures that a process gets a round number r only if the round $(r-1)$ has finished (cf. Lemma 4.1). The round number then is written to a shared data PRO_i (lines 16 and 17), where the data structure of PRO_i is described in Algorithm 5. These make the RMW procedure satisfy the requirement for using the LONGLIVEDCONSENSUS procedure (cf. Requirement 1).

After getting a round number $round_i$, p_i creates its own proposal for the long-lived consensus protocol in $round_i$. It finds one of the participating processes of the latest round (e.g p_k) and reads its result (e.g. $WINNER[k]$) (lines 4-9). The read value is checked to ensure that it is the result of round $(round_i - 1)$ (lines 10-14) (cf. Lemma 4.4). The result, which contains responses to functions that have been executed up to round $(round_i - 1)$, is copied to p_i's proposal PRO_i so that if $PRO_i.response[j] = res_k.response[j], \forall j$, the field $PRO_i.response[j]$ is kept unchanged. The same approach is used for the $toggle$ field of PRO_i (cf. the $Proposal$ data structure in Algorithm 5). Only responses/toggle-bits corresponding to the processes that have submitted a new function to FUN, are updated to new values (lines 19-23). This approach results in an important property of our RMW procedure:

Property 4.1: For any process p_i, if its current function f has been executed in a round r, the response to f in any process' buffer is kept unchanged until p_i submit a new function to $FUN[i]$.

Since p_i submits a new function only when making another invocation of the RMW procedure (line 2), this property implies that if a process p_i obtains a reference to a buffer containing the response to p_i's function f in a round r, it can later use this reference to get the correct response to its function f even if that buffer has been re-used for a proposal of later rounds $r' > r$.

After creating a proposal buf_i, an order of functions to be executed on the RMW object in round $round_i$, p_i uses the long-lived consensus object developed in Section 3 to achieve an agreement among processes in $round_i$ (line 26). If p_i's function has been executed in the agreement, p_i atomically writes the agreement $winner$ and its round $round_i$ to $WINNER[i]$ and $COU[i]$ (line 38) before returning the response $winner.response[i]$ (line 42).

Each process p_i has two buffers in order to achieve recycling: the *working* buffer $PRO_i[curBuf]$ is used to create proposal data and the *shared* buffer $PRO_i[\neg curBuf]$ is used to share the proposal data that has been chosen by the consensus protocol. If processes agree on p_i's proposal, p_i prepares the working buffer for the next round by triggering its $curBuf$ bit (line 40).

One of the biggest challenges in designing the RMW object using M_ASSIGNMENT operations is that proposal data cannot be stored in one register whereas the M_ASSIGNMENT operation can atomically write M values to M memory locations only if the values each can be stored in one register. Our RMW object overcomes the problem by ensuring Property 4.1 and using *references* to proposal data, instead of proposal data, as inputs for the LONGLIVEDCONSENSUS procedure. The consensus procedure returns an agreed reference of a buffer containing a proposal. If the proposal contains a response

Algorithm 6 RMW(f: function) invoked by process p_i

1: $toggle_i \leftarrow \neg FUN[i].toggle$;
2: $FUN[i] \leftarrow \{f, toggle_i\}$;
3: **for** l in $1...2$ **do**
4: $cou_i \leftarrow$ FASTSCAN(COU); $round_i \leftarrow \max_{1 \le j \le N} cou_i[j] + 1$; Let k be an index such that $cou_i[k] = \max_{1 \le j \le N} cou_i[j]$.
5: **if** $WINNER[k] = \perp$ **then** // Initial round, no previous winner $\Rightarrow$ Compute p_i's proposal data
6: $buf_i \leftarrow \&PRO_i[curBuf]$; // use buf_i as the reference/address of p_i's working buffer $PRO_i[curBuf]$
7: $buf_i.round \leftarrow round_i$; $buf_i.owner \leftarrow i$; // Update fields $round$ and $owner$ of $PRO_i[curBuf]$.
8: **else** // There is a winner in the previous round.
9: $res_k \leftarrow copy(WINNER[k])$; // Copy (non-atomically) the RMW object to a local buffer res_k.
10: $action \leftarrow$ CHECKRESULT($res_k, cou_i[k]$); // Check the result res_k.
11: **if** $action = Done$ **then**
12: **return** $res_k.response[i]$; // $round_i$ has finished $\Rightarrow$ p_i returns.
13: **else if** $action = Retry$ **then**
14: **continue**; // $round_i$ has finished but $FUN[i]$ of $round_i$ hasn't been executed. Retry.
15: **end if**
// $round_i = res_k.round + 1 \Rightarrow$ Compute p_i's proposal data
16: $buf_i \leftarrow \&PRO_i[curBuf]$; // use buf_i as the reference/address of p_i's working buffer $PRO_i[curBuf]$
17: $buf_i \leftarrow copy(res_k)$; $buf_i.round \leftarrow round_i$; $buf_i.owner \leftarrow i$; // Copy (non-atomically) the local buffer res_k to $PRO_i[curBuf]$ and update fields $round$ and $owner$ of the copy.
18: **end if**
// Apply proposed functions on the local copy.
19: $fun_i \leftarrow$ FASTSCAN(FUN);
20: **for** j in $1...N$ **do**
21: **if** $fun_i[j].toggle \neq buf_i.toggle[j]$ **then**
22: $buf_i.toggle[j] \leftarrow fun_i[j].toggle$;
23: $buf_i.response[j] \leftarrow buf_i.value$; $buf_i.value \leftarrow fun_i[j](buf_i.value)$;
24: **end if**
25: **end for**
// long-lived consensus
26: $winner \leftarrow$ LONGLIVEDCONSENSUS(buf_i); // p_i's round number is stored in $buf_i.round$.
27: **if** $winner = \perp$ **then** // $round_i$ had finished and a new round has started
28: **if** $l = 2$ **then** // p_i's 2^{nd} try and $round_i$ finished $\Rightarrow response_i$ must be ready
29: $cou_i \leftarrow$ FASTSCAN(COU); Let k be an index such that $cou_i[k] = \max_{1 \le j \le N} cou_i[j]$.
30: $res_k \leftarrow WINNER[k]$;
31: **return** $res_k.response[i]$;
32: **else**
33: **continue**;
34: **end if**
35: **else if** $winner.toggle[i] \neq toggle_i$ **then**
36: **continue**; // $winner$ didn't execute $FUN[i]$ $\Rightarrow$ Retry one more round
37: **else**
38: M_ASSIGNMENT($\{WINNER[i], COU[i]\}, \{winner, round_i\}$); // Atomic 2-register assignment
39: **if** $winner.owner = i$ **then**
40: $BUF[i].curBuf \leftarrow \neg BUF[i].curBuf$; // p_i is the winner $\Rightarrow$ prepare a buffer for the next round
41: **end if**
42: **return** $winner.response[i]$;
43: **end if**
44: **end for**

to p_i's function, the response will be kept unchanged until p_i gets the response and returns from the RMW procedure according to Property 4.1. Therefore, processes still achieve an agreed order of their functions executed on the RMW object although the buffer may be re-used for later rounds.

4.1 Correctness proofs

Lemma 4.1: If no process has finished a round r, no process can obtain a round number $r' \geq (r+1)$.

Proof: Since $round_i$ finishes as soon as a process p_i writes $round_i$ to $COU[i]$ (cf. Definition 4.4), a process p_n obtain a round number $round_n = \max_{1 \leq k \leq N} COU[k] + 1$ (Algorithm 6, line 4) only if the round $round_n - 1$ has finished. □

Lemma 4.2: In the CHECKRESULT procedure, the value res_k used from line 7C is a correct copy of $p_{res_k.owner}$'s shared buffer.

Proof: It may happen that when p_i makes a copy res_k of buffer $WINNER[k]$ (line 9 in RMW), the buffer has been re-used (or has become the working buffer) for a later round since p_i found k (line 4). Note that $WINNER[k]$ contains only a reference to the buffer containing proposal data due to M_ASSIGNMENT's register-size restriction. We prove the lemma by contradiction.

Assume that this scenario happens. Let $round_a$ be the round at which $WINNER[k]$ is updated with a reference to $round_a$'s winning buffer *Buffer*$_1$ that is being copied by p_i at line 9. Since i) $WINNER[k]$ and $COU[k]$ are updated in one atomic step using M_ASSIGNMENT (line 38), ii) $COU[k]$ is read to $cou_i[k]$ (line 4) before $WINNER[k]$ is read (line 9) and iii) $COU[k]$ is always increasing, $cou_i[k] \leq round_a$ (or $round_k \leq round_a$)

Let p_o be the owner of *Buffer*$_1$. Let $round_{owner}$ be the value of $COU[o]$ read by p_i at line 1C in CHECKRESULT. Since *Buffer*$_1$ has been re-used as a *working* buffer due to the hypothesis, there exists a smallest round $round_e$, $round_a < round_e \leq round_{owner}$, in which p_o was again the winner (line 40 is the only place p_o switches its *working* and *shared* buffers). It follows that $round_{owner} \geq round_e > round_a \geq round_k$, which makes the CHECKRESULT procedure return earlier (lines 3C and 5C), a contradiction to the hypothesis that this res_k value is used from line 7C. □

Lemma 4.3: The CHECKRESULT procedure returns OK only if $res_k.round = round_i - 1$.

Proof: Due to Lemma 4.2, res_k from line 7C is the result of the latest round that p_k has finished at the time p_i reads that value (line 9 in RMW). That round number is recorded in $res_k.round$ (line 17 in RMW). Since i) at line 12C in CHECKRESULT, $round_i > res_k.round$ (otherwise, the procedure returned at line 8C or 10C) and ii) $res_k.round \geq cou_i[k]$ (since res_k is read after cou_i and the round number is always increasing) and iii) $cou_i[k] = (round_i - 1)$ (line 4 in RMW), we have $round_i > res_k.round \geq (round_i - 1)$. Therefore, $res_k.round = (round_i - 1)$ at line 12C. □

Algorithm 7 CheckResult(res_k: reference; $round_k$: integer) invoked by process p_i

Output: OK, Done or Retry.
1C: $round_{winner} \leftarrow COU[res_k.owner]$;
2C: **if** $round_{winner} \neq round_k$ and $res_k.toggle[i] = toggle_i$ **then**
3C: **return** Done; // The winner has started a new round $\Rightarrow round_i$ had finished
4C: **else if** $round_{winner} \neq round_k$ and $res_k.toggle[i] \neq toggle_i$ **then**
5C: **return** Retry; // $round_i$ has finished but $FUN[i]$ of $round_i$ hasn't been executed. Retry.
6C: **end if**
// res_k is a *correct* copy of $p_{res_k.owner}$'s shared buffer.
7C: **if** $round_i \leq res_k.round$ and $res_k.toggle[i] = toggle_i$ **then**
8C: **return** Done; // $round_i$ has finished and $FUN[i]$ of $round_i$ has been executed. Done.
9C: **else if** $round_i \leq res_k.round$ and $res_k.toggle[i] \neq toggle$ **then**
10C: **return** Retry; // $round_i$ has finished, but $FUN[i]$ of $round_i$ hasn't been executed. Retry.
11C: **end if**
12C: **return** OK;

Lemma 4.4: The value res_k used to make p_i's proposal in round $round_i$ (line 17, Algorithm 6) is the result of round $(round_i - 1)$.

Proof: Since the CHECKRESULT procedure does not returned *Done* nor *Retry* only if $res_k.round = (round_i - 1)$ (Lemma 4.3), the value res_k used from line 17 in RMW satisfies $res_k.round = (round_i - 1)$ (otherwise the RMW procedure returned or retried earlier at line 12 or line 14, respectively). That means res_k used from line 17 is the result of round $(round_i - 1)$. □

Lemma 4.5: After a process p_i retries at line 14, 33 or 36 in Algorithm 6, p_i's function $FUN[i]$ will be executed by the winner of the next round at the latest.

Proof: Since i) p_i declares its latest function in $FUN[i]$ before $round_i$ finishes (lines 2 and 4) and ii) processes obtain the round number $(round_i + 1)$ only if $round_i$ has finished (cf. Lemma 4.1), processes participating in round $(round_i + 1)$ will definitely observe p_i's function when scanning FUN at line 19. The winner of round $(round_i+1)$ will realize that $FUN[i]$ has not been executed (line 21) since res_k is the result of round $round_i$ due to Lemma 4.4. Hence, $FUN[i]$ will be definitely executed by the winner of round $round_i + 1$. □

Therefore, if p_i's function has not been executed by the winner of $round_i$ and subsequently p_i retries and participates in a round $round_j \geq round_i + 1$, p_i will get the response to its function in $round_j$.

Lemma 4.6: Every process p_i will return with the response to its function after at most 2 iterations (line 3, Algorithm 6).

Proof: From Lemma 4.5, p_i's function will be executed at the latest in the round $round_j$ in which p_i participates during its second try. If p_i returns at line 12 or 42, the returned value is the response to its function due to Property 4.1. However, it may happens that $round_j$ has finished just before the invocation of the LONGLIVEDCONSENSUS procedure (line 26), making the procedure returns $\perp$ (line 27). In this case, p_i scans COU to get the result res_k of a round $round_r \geq round_j$, and $res_k.response[i]$ contains the response to p_i's function due to Property 4.1 (lines 29-31). Therefore, p_i will return with the response to its function after executing at most 2 iterations. □

Lemma 4.7: The RMW procedure is linearizable.

Proof: Assume that RMW(f_i) is invoked by process p_i. Within each round, participating processes achieve an agreement on the order of their functions to be executed using the LONGLIVEDCONSENSUS procedure and thus the functions of the participating processes each takes effect at one point within the execution of that round.

On the other hand, a function f_i that has been executed in a round will never be executed in later rounds during RMW(f_i). Indeed, assume that a function f_i is executed twice at round r_a by process p_j and at round $r_b, b > a$, by process p_l. Since f_i is executed at round r_a by p_j, p_j records $FUN[i].toggle$ in its proposal buf_j (i.e. $buf_j.toggle[i] = FUN[i].toggle$, line 22), which is the result of round r_a. Since f_i is executed again at round $r_b, b > a$, by p_l, p_l's variable $res_k.toggle[i]$ must be different from $FUN[i].toggle$ (lines 17, 19 and 21). Note that $FUN[i]$ is updated to $\{f_i, toggle_i\}$ only once in RMW(f_i) by its unique owner/process p_i (line 2). However, since p_l's res_k is the result of round $r_b - 1$ (due to Lemma 4.4) and $(r_b - 1) \geq r_a$ (due to hypothesis), it follows that $res_k.toggle[i] = buf_j.toggle[i]$ (due to Property 4.1). Since $buf_j.toggle[i] = FUN[i].toggle$, it follows that $res_k.toggle[i] = FUN[i].toggle$, a contradiction.

Therefore, there is a unique point in the whole execution (including many rounds) at which the function f takes effect. Since p_i doesn't invoke another RMW(f') before its previous RMW(f) has been completed, the unique point is the linearization point of the RMW(f). □

Lemma 4.8: The RMW procedure is a wait-free read-modify-write operation with the time complexity of $O(N)$.

Proof: Since the time complexity of LONGLIVEDCONSENSUS is $O(N)$ (Lemma 3.5) and RMW returns after at most two iterations of its for-loop (Lemma 4.6), the time complexity of RMW is $O(N)$. This also implies that RMW is wait-free. □

Lemma 4.9: The space complexity of the wait-free RMW object is $O(N^2)$, the optimal.

Proof: From the set of variables used to construct the RMW object (cf. Algorithm 5), we see that the $Proposal$ record has space complexity $O(N)$ and thus the space complexity of the BUF array is $O(N^2)$. Since the space complexity of the LONGLIVEDCONSENSUS procedure, which is used in the RMW procedure (line 26, Algorithm 6), is also $O(N^2)$ (cf. Lemma 3.8), the space complexity of the RMW object is $O(N^2)$.

On the other hand, any *general* wait-free RMW object (i.e. there is no restriction on function f) for N processes can be used as a building block to construct a wait-free (short-lived) consensus protocol for N processes with space complexity $O(1)$ (cf. the corresponding function f for the consensus protocol in Algorithm 8). Therefore, the space complexity of general wait-free RMW objects using only the M_ASSIGNMENT operation and read/write registers is $\Omega(N^2)$ due to Lemma 3.7. This means the space complexity $O(N^2)$ of the new wait-free RMW object (Algorithm 6) is optimal. □

Algorithm 8 Function F($agreement$) invoked by process p_i

Input: $agreement$ must be initialized to $\perp$ before the consensus protocol starts.

```
if agreement = ⊥ then
    agreement ← p_i's proposal;
    return p_i's proposal;
else
    return agreement;
end if
```

5 $(2M-3)$-RESILIENT READ-MODIFY-WRITE OBJECTS FOR ARBITRARY N

In this section, we present a $(2M-3)$-resilient RMW object for an arbitrary number N of processes using M_ASSIGNMENT operations. Since the operation has consensus number $(2M-2)$, we cannot construct any objects that tolerate more than $(2M-3)$ faulty processes using only the M_ASSIGNMENT operation and read/write registers [19].

Let $D = (2M-2)$ and, without loss of generality, assume that $N = D^{\mathcal{K}}$, where $\mathcal{K}$ is an integer. The idea is to construct a balanced tree with degree of D. Processes start from the leaves at level $\mathcal{K}$ and climb up to the first level of the tree, the level just below the root. When visiting a node at level $i, 2 \leq i \leq \mathcal{K}$, a process p_i calls the wait-free LONGLIVEDCONSENSUS procedure (cf. Section 3) for its D sibling processes/nodes to find an agreement on which process will be their representative that will climb up to the next higher level.

The representative process of p_i's D siblings at level l will invoke the wait-free LONGLIVEDCONSENSUS procedure for its D siblings at level $(l+1)$ and so on until the representative reaches level 1 of the tree at which there are exact D nodes. At this level, the D processes/nodes invoke the wait-free RMW procedure for D processes (cf. Section 4).

Fig. 4 illustrates the structure of the $(2M-3)$-resilient object. Each ellipse with label $(2M-2)$ represents a group of $(2M-2)$ processes/nodes and each edge with label *WF LLC* represents the representative of a group, which is chosen using the wait-free LONGLIVEDCONSENSUS procedure. The ellipse with label *WF RMW* at level 1 represents the group of $(2M-2)$ representatives that invoke the wait-free RMW procedure.

Processes that are not chosen to be the representative stop climbing the tree and repeatedly check the final result until their function is executed. After that they return with the corresponding response.

Particularly, a process p_i that wants to execute a function f on the resilient RMW object invokes the RESILIENTRMW procedure with f as its parameter (cf. Algorithm 9). The process checks whether it successfully climbs up to level 1 by calling the CANDIDATE procedure (line 3R and Algorithm 10) and if so, it invokes the wait-free RMW procedure for $(2M-2)$ siblings at level 1 (line 4R). Otherwise, p_i repeatedly reads the result to check if its function has been executed as in the RMW procedure (lines 8R, 9R and 13R). In order to reduce

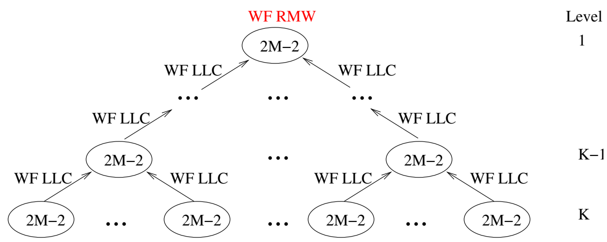

Fig. 4. The structure of $(2M-3)$-resilient RMW objects for arbitrary N

Algorithm 9 RESILIENTRMW(f: function) invoked by process p_i

1R: $toggle_i \leftarrow \neg FUN[i].toggle$
2R: $FUN[i] \leftarrow \{f, toggle_i\}$
3R: **if** CANDIDATE(i) = **true then**
4R: **return** RMW(f); // Wait-free read-modify-write object for $2M-2$ candidate processes
5R: **else**
6R: // Repeatedly check results with exponential backoff
7R: **repeat**
8R: $cou_i \leftarrow$ M_SCAN(COU); Let k be an index such that $cou_i[k] = \max_{1\leq j\leq N} cou_i[j]$;
9R: $result \leftarrow copy(WINNER[k])$;
10R: **if** $result.toggle[i] \neq toggle_i$ **then**
11R: Backoff before checking again;
12R: **end if**
13R: **until** $result.toggle[i] = toggle_i$
14R: **return** $result.response[i]$;
15R: **end if**

the contention level on the shared variables COU and $WINNER$, RESILIENTRMW delays for a while between two consecutive reads using the backoff mechanism [28].

Similar to invoking the RMW procedure, when invoking the CANDIDATE procedure, a process p_i first scans all N elements of COU to find the most recent round number $round_i$, the round it will belong to (line 1C). Since a round r finishes when r is recorded in a variable $COU[j]$ by a process p_j executing the RMW procedure at level 1 (cf. Definition 4.4), p_i gets a round number r only if the round $(r-1)$ has finished. The round number then is written to buf_i (or $PRO_i[currBuf]$) (line 2C). These make the CANDIDATE procedure satisfy the requirement for using the LONGLIVEDCONSENSUS procedure (cf. Requirement 1). At each intermediate level l on the path to the first level, p_i invokes LONGLIVEDCONSENSUS to achieve an agreement among its $(2M-2)$ siblings on their representative for the next higher level (line 4C). Process p_i will stop climbing up as soon as it is not chosen as a representative (line 6C).

The RMW procedure used in the RESILIENTRMW procedure is the same as the RMW procedure in previous section except that i) RMW doesn't initialize $FUN[i]$ since $FUN[i]$ is initialized at line 2R and ii) the FASTSCAN function, which takes a snapshot of $2M$ registers using M_READ and M_ASSIGNMENT operations with time complexity $O(1)$, is replaced by M_SCAN that takes a snapshot of arbitrary N registers using M_READ and M_ASSIGNMENT operations with time complexity of $O((\frac{N}{M})^2)$ [27]. This leads to the following lemma:

Algorithm 10 CANDIDATE(i: index) invoked by process p_i

1C: $cou_i \leftarrow$ M_SCAN(COU); $round_i \leftarrow \max_{1\leq j\leq N} cou_i[j] + 1$;
2C: $buf_i.round \leftarrow round_i$; $buf_i.owner \leftarrow i$;
3C: **for** $l = \mathcal{K}$ to 2 **do**
4C: $winner \leftarrow$ LONGLIVEDCONSENSUS$^l(buf_i)$; // Achieve an agreement among p_i's D siblings at level l on who is their representative. Return the ID of the winning process
5C: **if** $winner = \perp$ or $winner \neq i$ **then**
6C: **return false**;
7C: **end if**
8C: **end for**
9C: **return true**;

Lemma 5.1: For the correct processes[7] that execute RMW (line 4R in Algorithm 9), the time complexity of their RESILIENTRMW is $O(N^2)$ if M is a constant and is $O(N)$ if the ratio $\frac{N}{M}$ is a constant.

Proof: The time complexity of RMW (for D processes) using M_SCAN with time complexity $O((\frac{N}{M})^2)$ is $O((\frac{N}{M})^2 + D)$, where $D = 2M-2$. Since CANDIDATE uses M_SCAN (line 1C) and invokes LONGLIVEDCONSENSUS (for D processes) with time complexity $O(D)$ at each of $\log_D N$ levels (line 4C), the time complexity of CANDIDATE is $O((\frac{N}{M})^2 + D\log_D N)$. Therefore, the time complexity of RESILIENTRMW in this case is $O((\frac{N}{M})^2 + D + D\log_D N)$. If M is a constant, the time complexity becomes $O(N^2)$. If $\frac{N}{M} = \alpha$, where α is a constant, the time complexity becomes $O(N)$. □

Lemma 5.2: The ResilientRMW object is $(2M-3)$ resilient for an arbitrary number N of processes.

Proof: We will prove that correct processes always return with a response to its function if at most $(2M-3)$ processes, which are accessing the object, fail (cf. the t-resilient model in Section 2).

Since at most $(2M-3)$ processes fail, at least one of $(2M-2)$ processes at level 1 is correct and successfully executes the RMW procedure, ensuring that the final result exists. Due to Property 4.1, the responses to processes functions in the final result are kept unchanged until the processes submit their new function. Therefore, the response returned at line 4R or 14R is the response to p_i's function. That means every *correct* process p_i will eventually get its response and return via either repeatedly checking the final result (line 14R) or executing the wait-free RMW procedure at level 1 (line 4R). □

6 CONCLUSIONS

In this paper, based on the intrinsic features of emerging media/graphics processing unit (GPU) architectures we have generalized the architectures to an abstract model of an MIMD chip with multiple SIMD cores sharing a memory. For this general model, which makes no assumption on the existence of strong synchronization primitives such as *test-and-set* and *compare-and-swap*, we have developed a wait-free *long-lived* consensus (LLC) object for $N = (2M-2)$ cores, where M is at most

7. *Correct* processes are processes that do not crash in the object execution.

the number of hardware threads on each core. The time complexity of the new consensus algorithm is $O(N)$, which is better than the time complexity $O(N^2)$ of the well-known *short-lived* consensus algorithm on the same setting [6]. Using the long-lived consensus object, we have developed a wait-free long-lived *read-modify-write* (RMW) object for $N = (2M - 2)$ with time complexity $O(N)$. Both the LLC object and the RMW object have the optimal space complexity $O(N^2)$. In the case $N > (2M - 2)$, we have developed a $(2M - 3)$-resilient RMW object for an arbitrary number N of cores.

The results presented in this paper provide a starting point to bridge the gap between the lack of strong synchronization primitives in several GPUs and the need for strong synchronization mechanisms in parallel applications. The results show that wait-free programming is possible for GPUs without hardware synchronization primitives such as *test-and-set* and *compare-and-swap*, extending the set of parallel applications that can utilize the ubiquitous and powerful computational hardware.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful and thorough comments on the earlier version of this paper. Otto Anshus's work was supported by the Research Council of Norway as part of projects SHARE and Display Wall (grant numbers 159936/V30 and 155550/420) and by the Tromsø Research Foundation. Philippas Tsigas's work was partially supported by the EU as part of FP7 Project PEPPHER (grant number 248481) and the Swedish Foundation for Strategic Research as part of the project RIT-10-0033.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] A. Munshi, *The OpenCL Specification, version 1.0.48*. Khronos OpenCL Working Group, 2009.

[3] *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 3.1*. NVIDIA Corporation, 2010.

[4] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *Proc. of the IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, 2004, p. 177.

[5] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Proc. of the ACM Symp. on Graphics Hardware (GH)*, 2007, pp. 55–64.

[6] M. Herlihy, "Wait-free synchronization," *ACM Transaction on Programming and Systems*, vol. 11, no. 1, pp. 124–149, 1991.

[7] Q. Hou, K. Zhou, and B. Guo, "Bsgp: bulk-synchronous gpu programming," in *ACM SIGGRAPH*, 2008, pp. 1–12.

[8] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," in *Proc. of the IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2010, pp. 1–12.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[10] E. Borowsky and E. Gafni, "Generalized flp impossibility result for t-resilient asynchronous computations," in *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 91–100.

[11] P. H. Ha, P. Tsigas, and O. J. Anshus, "The synchronization power of coalesced memory accesses," in *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, 2008, pp. 320–334.

[12] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Trans. Comput. Syst.*, vol. 23, no. 2, pp. 146–196, 2005.

[13] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[14] R. De Prisco, B. Lampson, and N. Lynch, "Revisiting the paxos algorithm," *Theor. Comput. Sci.*, vol. 243, no. 1-2, pp. 35–91, 2000.

[15] T.-M. S. Hagen, P. H. Ha, and O. J. Anshus, "Experimental fault-tolerant synchronization for reliable computation on graphics processors," in *Proc. of the Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008, p. to appear, http://www.cs.uit.no/∼phuong/para08_140.pdf.

[16] D. Pham and et.al., "The design and implementation of a first-generation cell processor," in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005, pp. 184–185.

[17] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc., 2004.

[18] P. Micikevicius, *Personal communication*. Nvidia, 2008.

[19] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg, "Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus," *SIAM Journal on Computing*, vol. 34, no. 2, pp. 333–357, 2005.

[20] L. Lamport, "Concurrent reading and writing," *Commun. ACM*, vol. 20, no. 11, pp. 806–811, 1977.

[21] G. L. Peterson, "Concurrent reading while writing," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 46–55, 1983.

[22] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, pp. 77–97, 1987.

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[24] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient synchronization of multiprocessors with shared memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 4, pp. 579–601, 1988.

[25] M. Herlihy, "Randomized wait-free concurrent objects (extended abstract)," in *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, 1991, pp. 11–21.

[26] C. Dwork and M. Herlihy, "Bounded round number," in *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, 1993, pp. 53–64.

[27] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *J. ACM*, vol. 40, no. 4, pp. 873–890, 1993.

[28] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Procs. of the Annual Intl. Symp. on Computer Architecture*, 1989, pp. 396–406.

Phuong Hoai Ha received the Ph.D. degree from the Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. Currently, he is an associate professor in the Department of Computer Science, University of Tromsø, Norway. His research interests include parallel/distributed computing and systems with the focus on scalable concurrency, efficient memory access mechanisms, concurrent data structures and algorithms, and parallel programming (www.cs.uit.no/∼phuong).

Philippas Tsigas received the BSc degree in mathematics and the PhD degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present, he is a professor in the Department of Computing Science at Chalmers University of Technology, Sweden. His research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing, and information visualization (www.cs.chalmers.se/∼tsigas).

Otto J. Anshus is a professor of computer science at the University of Tromsø. His research interests include operating systems, parallel and distributed architectures and systems, scalable display systems, data-intensive computing, high-resolution visualizations, and human-computer interfaces. He is a member of the IEEE Computer Society, the ACM, and the Norwegian Computer Society. Contact him at otto.anshus@uit.no.