Wait-free Programming for General Purpose Computations on Graphics Processors

Phuong Hoai Ha
University of Tromsø
Department of Computer Science
Faculty of Science, N-9037 Tromsø, Norway
phuong@cs.uit.no

Philippas Tsigas
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg, Sweden
tsigas@cs.chalmers.se

Otto J. Anshus
University of Tromsø
Department of Computer Science
Faculty of Science, N-9037 Tromsø, Norway
otto@cs.uit.no

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. Recent GPUs feature the single-program multiple-data (SPMD) multicore architecture instead of the single-instruction multiple-data (SIMD). However, unlike CPUs, GPUs devote their transistors mainly to data processing rather than data caching and flow control, and consequently most of the powerful GPUs with many cores do not support any synchronization mechanisms between their cores. This prevents GPUs from being deployed more widely for general-purpose computing.

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

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 [16]. Since GPUs are specialized for computation-intensive highly-parallel applications (e.g. graphics rendering), unlike CPUs, GPUs devote more transistors to data processing rather than data caching and flow control. Current GPUs are capable of about ten times as many GFLOPS[1] as current CPUs. GPU computational power doubles every ten months (surpassing the Moore's Law for traditional microprocessors) whereas CPU computational power doubles every seventeen months. These facts have motivated researchers to utilize the ubiquitous and powerful GPU for general-purpose computing such as physics simulations, data mining and signal processing [16]. Moreover, unlike previous GPU architectures, which are single-instruction multiple-data (SIMD), recent GPU architectures (e.g. Compute Unified Device Architecture (CUDA) [1]) are single-program multiple-data (SPMD). The latter consists of multiple SIMD multiprocessors of which each, at the same time, can 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, the recent GPU architecture also creates challenges on synchronization between its SIMD multiprocessors (or SIMD cores). Since the GPU is designed to de-

[1] Giga FLoating point Operations Per Second

vote transistors to computation rather than data caching and flow control, most of the current powerful GPUs with many cores (e.g. NVIDIA Tesla series with up to 64 cores and GeForce 8800 series with 16 cores) do not support strong synchronization primitives like *test-and-set* and *compare-and-swap* [1]. Due to lack of synchronization mechanisms between the SIMD cores, the SIMD cores cannot safely communicate with each other through shared memory [1]. On the other hand, most of the parallel applications need some synchronization mechanism to synchronize their concurrent processes. The fact prevents the GPU from being deployed more widely.

The paper aims at bridging the gap between the lack of synchronization mechanisms in the GPU architecture and the need of synchronization mechanisms in parallel applications. Based on the intrinsic features of recent GPU architectures, we first generalize the architectures to an abstract model of a chip with multiple SIMD cores sharing a memory (cf. Section 2). Each core can process M threads (in a SIMD manner) in one clock cycle. Each thread of a core accesses the shared memory using (atomic) read/write operations. Then, we construct wait-free and t-resilient synchronization objects [4, 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 [13, 15, 19, 20].

We observe that due to SIMD architecture each SIMD core with M hardware threads can read/write M memory locations in one atomic step. Using the M-register read/write operations we construct a wait-free (long-lived) read-modify-write (RMW) objects in the case the number N of cores is not greater than $(2M-2)$ (cf. Section 4). In the case $N > (2M-2)$, we construct $(2M-3)$-resilient RMW objects using only the M-register operations and read/write registers (cf. Section 5). It has been proved that $(2M-3)$ is the maximum number of crash failures that a system with M-register assignments and read/write registers can tolerate while ensuring consensus for correct processes[2] [5, 10]. Therefore, from a fault-tolerant point of view, these wait-free/resilient objects are the best we can achieve. To the best of our knowledge, research on constructing wait-free and $(2M-3)$-resilient long-lived RMW objects using only M-register read/write operations and read/write registers has not been reported previously.

In order to construct the wait-free/resilient long-lived RMW objects for this model, there are challenges to be handled. First, unlike *short-lived* wait-free/resilient consensus objects [5, 10] in which the object variables are used once during the object life-time, *long-lived* wait-free/resilient consensus objects must allow processes to re-use the object variables so as to keep the object size bounded. This implies that the long-lived objects must include a wait-free/resilient memory management mechanism [11, 14] inside themselves. Another challenge is that processes concurrently accessing a wait-free/resilient long-lived *read-modify-write* object must agree on a proposal that contains *all* responses to the object operations suggested by the processes. Therefore, unlike the process proposal in the consensus object, the process proposal in the RMW object is unable 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 object must handle the proposal-size issue while tolerating the same number of crash failures $(2M-3)$ as the consensus object.

The main contribution of this paper is to design a set of universal synchronization objects capable of empowering the programmer with the necessary and sufficient tools for wait-free programming on graphics processors. The technical contributions of this paper are threefold:

- We develop a wait-free *long-lived* consensus object for $N = (2M-2)$ processes using only M-register read/write operations and read/write registers. The consensus algorithm has time complexity $O(N)$ (cf. Section 3). The time complexity is better than the time complexity $O(N^2)$ of the well-known *short-lived* consensus algorithm using M-register assignments [10]. The short-lived consensus algorithm needs to construct a directed graph of processes in the second phase, leading to the time complexity $O(N^2)$.

- We develop a wait-free long-lived RMW object for $N = (2M-2)$ processes using only M-register read/write operations and read/write registers. Accesses to the RMW object have time complexity $O(N)$ (cf. Section 4). This result implies that it is possible to construct wait-free synchronization mechanisms for GPUs without the need of strong synchronization primitives in hardware.

- We develop a $(2M-3)$-resilient long-lived RMW object for an arbitrary number N of processes using only M-register read/write operations and read/write registers (cf. Section 5).

The rest of this paper is organized as follows. Section 2 presents a general model of a 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 RMW object for $N = (2M-2)$ processes. Section 5 presents the $(2M-3)$-resilient RMW object for an arbitrary number N of processes. Section 6 suggests a solution to make the round number bounded. Finally, Section 7 concludes this paper.

[2] *Correct* processes are processes that do not crash in the execution.

2 The Model

Inspired by emerging media/graphics processing unit architectures like CUDA [1] and Cell BE [18], the abstract system model we consider in this paper is illustrated in Fig. 1. The model consists of N SIMD-cores sharing a shared memory and each core can process M threads (in a SIMD manner) in one clock cycle. For instance, the GeForce 8800GTX graphics processor, which is the flagship of the CUDA architecture family, has 16 SIMD-cores/SIMD-multiprocessors, each of which processes up to 16 concurrent threads in one clock cycle.

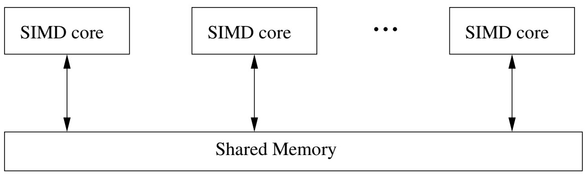

Figure 1. The abstract model of a chip with multiple SIMD-cores

Since powerful media/graphics processing units with many cores (e.g. NVIDIA Tesla series with up to 64 cores and GeForce 8800 series with 16 cores) do not support strong synchronization primitives like *test-and-set* and *compare-and-swap* [1], we make no assumption on the existence of such strong synchronization primitives in this model. In this model, each of the M threads of one SIMD core can read/write one memory location in one atomic step. Due to SIMD architecture, each SIMD core can read/write M different memory locations in one atomic step or, in order words, each SIMD core can execute M_READ and M_ASSIGNMENT (atomic) operations.

Different cores can concurrently execute different user programs and a process, which sequentially executes instructions of a program on one core, can crash due to the program errors. The failure category considered in this model is the crash failure: a failed process cannot take another 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 [5].

Terminology Synchronization objects are conventionally classified by *consensus number*, the maximum number of processes for which the object can solve a consensus problem [10]. An n-*consensus object* allows n processes to propose their values and subsequently returns only one of these values to *all* the n processes. A *short-lived* (resp. *long-lived*) consensus object is a consensus object in which the object variables are used once (resp. many times) during the object life-time. An object implementation is *wait-free* if any process can complete any operation on the object in a finite number of steps regardless of the execution speeds of other processes [10, 12, 17]. An object implementation is *t-resilient* if non-faulty processes can complete their operations as long as no more than t processes fail [6, 8].

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

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 3.1. The problem is to construct a long-lived object that guarantees consensus among processes with the same consensus number (or processes within the same round) using M_ASSIGNMENT operation. Since i) the adversary can arrange all N processes to be in the same round and ii) the M_ASSIGNMENT operation has consensus number $(2M - 2)$, we cannot construct any wait-free objects that guarantee consensus for more than $(2M-2)$ processes using only the operation and read/write registers [10], 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 3.1. *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, and*
- *processes declare their current round number in shared variables before participating in a consensus protocol.*

At this moment, round numbers are assumed to be unbounded for the sake of simplicity. Solutions to make the round numbers bounded are presented in Section 6.

In the rest of this section, we presents a wait-free long-lived consensus (LLC) object for $N = (2M - 2)$ processes using M_ASSIGNMENT operations. The LLC object is developed from the short-lived consensus (SLC) object using M_ASSIGNMENT in [10]. The LLC object will be used to achieve an agreement among processes in the same 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.

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

---

$ROUND[1...N]$: contains current round numbers of N processes. $ROUND[i]$ is written by only process p_i and can be read by all N processes. $ROUND[i]$ must be set before p_i calls this LONGLIVEDCONSENSUS procedure.

$REG[\ ][\ ]$: 2-writer registers. $REG[i][j]$ can be written by processes p_i and p_j. For the sake of simplicity, we use a virtual array $2WR[1..M][1..M]$ that is mapped to REG of size $\frac{M(M-1)}{2}$ as follows

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

$1WR[1...M][0...1]$: 1-writer registers. $1WR[i]$ can be written by only process p_i.

Input: a unique proposal buf_i for p_i **and** p_i's round number $ROUND[i]$
Output: a proposal **or** $\perp$
1L: $gId \leftarrow \lfloor \frac{i}{M-1} \rfloor$ // 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 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** $\perp$ // pId's round had finished and a new round started
9L: **end if**
10L: **return** $winner$

The algorithm of the wait-free LLC object using M_ASSIGNMENT is presented in Algo. 1. Before a process p_i invokes the LONGLIVEDCONSENSUS procedure, p_i's round number must be declared in the shared variable $ROUND[i]$. 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.

A process p_i proposes its data by passing its proposal data to the procedure. Like the SLC object in [10], when the proposal data is unique for each process and can be stored in a register, the LLC object can work directly on the proposal data. However, when the proposal data is either not unique for each process or larger than the register size, which makes M_ASSIGNMENT no longer able to atomically write M proposal data, our LLC object works on the references to (or addresses of) the proposal data with the condition that processes allocate their own memory to contain their proposal data. In this case, applications using the LLC object must ensure that processes, after achieving an agreed reference ref, read the correct proposal data matching ref. Even though processes get the same reference ref via the LLC object, the data to which the reference refers may change, making processes get different data.

Like the SLC algorithm [10], 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).

Note that p_i's round number is unchanged when p_i is executing the LONGLIVEDCONSENSUS procedure. If p_i's round 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).

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

---

Output: $\perp$ **or** the proposal of the earliest process in p_i's round
1F: M_ASSIGNMENT($\{1WR[i][gId], 2WR[i][\alpha+1], \cdots, 2WR[i][\alpha+M-1]\}, \{buf_i, \cdots, buf_i\}$), where $\alpha = gId(M-1)$
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: $\{first, ref\} \leftarrow$ ORDERING$(first, k, gId)$ // Find the earliest process $first$ of p_i's group in p_i's round
5F: **if** $first = \perp$ **then**
6F: **return** $\perp$ // pId's round had finished and a new round started
7F: **end if**
8F: **end for**
9F: **return** ref // $first$'s proposal in p_i's round

The FIRSTAGREEMENT procedure (cf. Algo. 2) simply scans all members of p_i's group to find the earliest process using the ORDERING procedure (cf. Algo. 5). The ORDERING procedure receives as input two processes and returns the preceding one together with its proposal in p_i's round. Since the preceding order is transitive, the variable $first$ after the for-loop is the first process of p_i's group in p_i's round.

The SECONDAGREEMENT procedure (cf. Algo. 3) is an innovative improvement of the abstract idea in the SLC algorithm [10]. 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 the formers assignment precedes the latter's (or the former precedes the latter 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 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 of 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.

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

1S: M_ASSIGNMENT($\{1WR[i][\neg gId], 2WR[i][\beta + 1], \cdots, 2WR[i][\beta + M - 1]\}, \{first, \cdots, first\}$), where $\beta = (\neg gId)(M-1)$
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: $previous \leftarrow winner$
9S: $\{winner, ref\} \leftarrow$ ORDERING($winner, next, \neg w\_gId$)
10S: **if** $winner = \perp$ **then**
11S: **return** $\perp$ // pId's round had finished and a new round started
12S: **else if** $winner \neq previous$ **then**
13S: $w\_gId \leftarrow \neg w\_gId$ // $winner$ now belongs to the other group
14S: $next \leftarrow previous$
15S: **end if**
16S: $next \leftarrow$ 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 // The reference to $winner$'s proposal in round $round_i$

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. 2 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. 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 $p_{i+2}^0 \rightsquigarrow p_1^1$ and $p_{i+2}^0 \rightsquigarrow p_2^1$ for short) since $p_{i+2}^0 \rightsquigarrow p_3^1 \rightsquigarrow p_i^0$ 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.

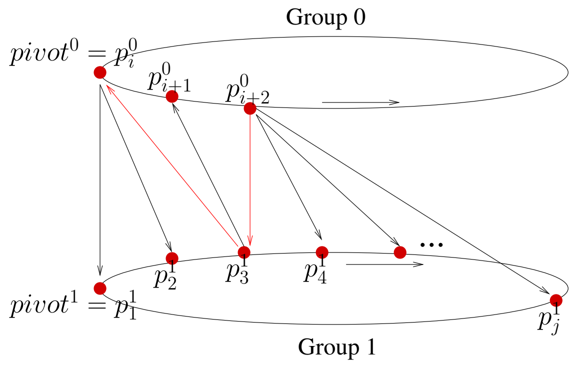

Figure 2. Illustration for the SECONDAGREEMENT **procedure, Algo. 3**

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 earlier process between two processes $winner$ and $next$ in the same round $round_i$ (cf. Lemma 3.3), the final winner $\mathcal{W}$ precedes all processes that are checked in the repeat-until loop (lines 7S - 17S). 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 round via 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.

□

We now show that values of shared variables (e.g. $2WR$ and $1WR$) used by the FIRSTAGREEMENT and SECONDAGREEMENT procedures belong to p_i's round, and thus the correctness of the LLC algorithm comes directly from the SLC correctness proof in [10]. The shared variables are read in the ORDERING procedure (Algo. 5). The procedure ensures that the values read from the shared variables belong to p_i's round by checking that the processes writing to the shared variables are in p_i's round both before and after reading the variables (lines 1O and 6O).

Particularly, p_i invokes the CHECKROUND procedure (cf. Algo. 4) to check whether processes $first$ and k are still in p_i's round before and after it reads the shared variables $2WR[first][k]$, $1WR[first][gId]$ and $1WR[k][gId]$ (lines 1O, 5O and 6O). The CHECKROUND$(first, k)$ procedure returns i) $\perp$ if p_i's round has finished (line 4H), ii) $first$ if process p_{first} is in p_i's round and p_k's round has finished (line 6H) or iii) p_i's round if both p_{first} and p_k are in p_i's round (line 8H). These result from the Lemma 3.2.

Lemma 3.2. *In the* CHECKROUND *procedure (Algo. 4), from line 5H,* $round_{first} = round_i$ *and* $round_k \leq round_{first}$.

Proof. Since i) processes' round numbers are increasing (cf. Requirement 3.1, item 1), ii) $first$ is initialized to i before p_i calls ORDERING (lines 2F in Algo. 2 and 2S in Algo. 3) and iii) $round_i$ is unchanged while p_i is executing LONGLIVEDCONSENSUS (cf. Requirement 3.1, items 1 and 3), $round_{first} \geq round_i$ in the CHECKROUND procedure. Therefore, from line 5H, $round_{first} = round_i$ and thus $round_k \leq (round_i =) \, round_{first}$ (otherwise, the procedure returned early at line 4H). □

Algorithm 4 CHECKROUND($first, k$: process index) invoked by process p_i

---

Output: $\perp$, $first$ **or** p_i's round number.
1H: $round_i \leftarrow ROUND[i]$ // Get p_i's round
2H: M_READ$(\{round_{first}, round_k\}, \{ROUND[first], ROUND[k]\})$
3H: **if** $round_i < round_{first}$ or $round_i < round_k$ **then**
4H: **return** $\perp$ // p_i's round has finished.
5H: **else if** $round_{first} > round_k$ **then**
6H: **return** $first$ // $round_i = round_{first}$ and $round_k$ has finished $\Rightarrow$ Ignore k.
7H: **else**
8H: **return** $round_k$ // $round_i = round_{first} = round_k \Rightarrow$ Return the round number
9H: **end if**

Algorithm 5 ORDERING($first, k$: index; gId: bit) invoked by process p_i

---

Output: $\{\perp,\perp\}$ or {index, proposal}
1O: $checkR_1 \leftarrow$ CHECKROUND$(first, k)$
2O: **if** $checkR_1 = \perp$ **then**
3O: **return** $\{\perp, \perp\}$ // $round_i$ has finished.
4O: **end if**
5O: $2wr_{first,k} \leftarrow 2WR[first][k]$; $1wr_{first} \leftarrow 1WR[first][gId]$; $1wr_k \leftarrow 1WR[k][gId]$
6O: $checkR_2 \leftarrow$ CHECKROUND$(first, k)$
7O: **if** $checkR_2 = \perp$ **then**
8O: **return** $\{\perp, \perp\}$
9O: **end if**
10O: **if** $checkR_1 = first$ or $checkR_2 = first$ **then**
11O: **return** $\{first, 1wr_{first}\}$ // $round_i = round_{first}$ and $round_k$ has finished $\Rightarrow$ Ignore k.
12O: **end if**
// $round_i = round_{first} = round_k$ and $2wr_{first,k}$, $1wr_{first}$ and $1wr_k$ are values in round $round_i$
13O: **if** $2wr_{first,k} = 1wr_k$ **then**
14O: **return** $\{first, 1wr_{first}\}$
15O: **else**
16O: **return** $\{k, 1wr_k\}$
17O: **end if**

Lemma 3.3. *The* ORDERING *procedure returns*

- $\perp$ *only if* p_i*'s round (or* $round_i$ *for short) has finished.*
- *a pair {index, proposal} in which proposal is* p_{index}*'s proposal in* p_i*'s round and* $index$ *is either the preceding between* $first$ *and* k *in the case that both are being in* p_i*'s round, or* $first$ *in the case that* $round_k$ *has finished and* $round_{first} = round_i$.

Proof. The first part of this lemma is clear from the ORDERING pseudocode. The procedure returns $\perp$ iff CHECKROUND (Algo. 4) returns $\perp$ (lines 3O and 8O). CHECKROUND returns $\perp$ only if $round_i$ has finished (line 4H).

Now we prove the second part of the lemma. The procedure returns a pair $\{index, proposal\}$ only at lines 11O, 14O and 16O. Since i) CHECKROUND is executed both before and after variable $1WR[first][gId]$ are read (lines 1O, 5O and 6O), ii) CHECKROUND doesn't return $\perp$ at lines 3O and 8O only if $round_{first} = round_i$ (cf. Lemma 3.2) and iii) $round_i$ is unchanged while p_i is executing LONGLIVEDCONSENSUS (cf. Requirement 3.1, items 1 and 3), the value $1wr_{first}$ returned at line 11O is the value of $1WR[first][gId]$ within $round_i$. Note that $1WR[first][gId]$ is written only by process p_{first}.

Similarly, the values $2wr_{first,k}$ and $1wr_k$ used from line 13O are the values of $2WR[first][k]$ and $1WR[k][gId]$ within $round_i$. Indeed, the fact that CHECKROUND doesn't return $\perp$ nor $first$ both before and after the values are read (lines 1O, 5O, 6O) ensures that the values are read within $round_i$ (cf. Algo. 4). Therefore, $1wr_{first}$ returned at lines 11O and 14O, and $1wr_k$ returned at line 16O are the proposals of processes $first$ and k in $round_i$.

We prove the last part of the lemma. It is clear from the ORDERING and CHECKROUND pseudocodes that the ORDERING procedure returns $first$ at line 11O only if $round_k$

has finished and $round_{first} = round_i$ (cf. line 6H, Algo. 4).

In the case $round_i = round_{first} = round_k$ (i.e from line 13O), since i) $first$ is initialized to i and ii) $1WR[i][gId]$ and $2WR[i][k]$ are *atomically* updated to p_i's proposal in $round_i$ before p_i calls ORDERING (lines 1F, 2F in Algo. 2 and 1S, 2S in Algo. 3), if $2wr_{first,k} = 1wr_k$, the process k has come after the process $first$ and overwritten $2WR[first][k]$. Therefore, $first$ is the preceding and is returned (line 14O). Otherwise, k is the preceding and is returned (line 16O). Note that the proposal data is unique for each process. □

Lemma 3.4. *The time complexity of the* LONGLIVEDCONSENSUS *procedure is* $O(N)$.

Proof. The time complexity of CHECKROUND is $O(1)$ and thus the time complexity of ORDERING is also $O(1)$. Since FISRTAGREEMENT scans $M = \frac{N+2}{2}$ processes of p_i's group to find the earliest one, its time complexity is $O(N)$. Since SECONDAGREEMENT checks at most N processes in the repeat-until loop (cf. Fig. 2), its time complexity is also $O(N)$. Therefore, the time complexity of LONGLIVEDCONSENSUS is $O(N)$. □

Lemma 3.5. *For any wait-free consensus protocols using only the* M_ASSIGNMENT *operation and read/write registers, the optimal space complexity is* $O(N^2)$.

Proof. It has been proved that in any wait-free consensus protocols using only the M_ASSIGNMENT operation and read/write registers, each pair of processes must have a register that is written only by those two processes (cf. the proof of Theorem 13 in [10]). Therefore, for N processes there must be at least $\frac{N(N-1)}{2}$ registers, which means that the optimal space complexity is $O(N^2)$. □

Lemma 3.6. *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. Algo. 1), the space complexity of the LLC object is obviously $O(N^2)$ due to array REG. Due to Lemma 3.5, 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 [10]. The idea is to divide the execution of the RMW object into consecutive rounds. Processes belonging to 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.*

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

Definition 4.3. *A round r is considered* finished *when all participating processes of this round achieve agreement. At that time, these participating processes* finish *the round r.*

Definition 4.4. *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.*

Algorithm 6 Data structures and variables used in Algo. 7

$Proposal$: **record** $owner, round, response[1..N], toggle[1..N], value$ **end**.
$BUF[1...N]$: **record** $curBuf, PRO[0..1]$ of $Proposal$ **end**. 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 $Proposal$: $WINNER[i]$ 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 $WINNER[i]$.
$FUN[1...N]$: $FUN[i]$ 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 $COU[i]$.
FASTSCAN(): scans a set of size less than $2M$ using the M-register read/write operations. Its time complexity and space complexity are $\Theta(1)$ [2]

Particularly, a process p_i, which wants to execute a function f on the RMW object, invokes the RMW procedure (Algo. 7) 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 18 and 20). 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 4.6).

In order to use the LONGLIVEDCONSENSUS procedure, each process needs to manage its own round number, which

Algorithm 7 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); $round_i \leftarrow \max_{1\leq j\leq N} cou_i[j] + 1$; Let k be an index such that $cou_i[k] = \max_{1\leq j\leq N} cou_i[j]$
5: $res_k \leftarrow copy(WINNER[k])$
6: **if** $COU[res_k.owner] \neq cou_i[k]$ and $res_k.toggle[i] = toggle_i$ **then**
7: **return** $res_k.response[i]$ // The winner has started a new round $\Rightarrow round_i$ has finished
8: **else if** $COU[res_k.owner] \neq cou_i[k]$ and $res_k.toggle[i] \neq toggle_i$ **then**
9: **continue** // $round_i$ has finished but $FUN[i]$ of $round_i$ hasn't been executed. Retry.
10: **end if**
11: **if** $round_i \leq res_k.round$ and $res_k.toggle[i] = toggle_i$ **then**
12: **return** $res_k.response[i]$ // $round_i$ has finished $\Rightarrow p_i$ returns.
13: **else if** $round_i \leq res_k.round$ and $res_k[i].toggle \neq toggle$ **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 $PRO[curBuf]$
17: $buf_i \leftarrow copy(res_k)$; $buf_i.round \leftarrow round_i$; $buf_i.owner \leftarrow i$
18: $fun_i \leftarrow$ FASTSCAN(FUN);
19: **for** j in 1...N **do**
20: **if** $fun_i[j].toggle \neq buf_i.toggle[j]$ **then**
21: $buf_i.toggle[j] \leftarrow fun_i[j].toggle$;
22: $buf_i.response[j] \leftarrow buf_i.value$; $buf_i.value \leftarrow fun_i[j](buf_i.value)$
23: **end if**
24: **end for**
// long-lived consensus
25: $winner \leftarrow$ LONGLIVEDCONSENSUS(buf_i)
26: **if** $winner = \perp$ **then**
27: **if** $l = 2$ **then**
28: $cou_i \leftarrow$ FASTSCAN(COU); Let k be an index such that $cou_i[k] = \max_{1\leq j\leq N} cou_i[j]$
29: $res_k \leftarrow copy(WINNER[k])$
30: **return** $res_k.response[i]$ // p_i's 2^{nd} try and $round_i$ finished $\Rightarrow response[i]$ must be ready.
31: **else**
32: **continue** // $round_i$ was finished and a new round has started
33: **end if**
34: **else if** $winner.toggle[i] \neq toggle_i$ **then**
35: **continue** // $winner$ didn't execute $FUN[i] \Rightarrow$ Retry one more round
36: **else**
37: M_ASSIGNMENT($\{WINNER[i], COU[i]\}, \{winner, round_i\}$)
38: **if** $winner.owner = i$ **then**
39: $BUF[i].curBuf \leftarrow \neg BUF[i].curBuf$ // p_i is the winner $\Rightarrow$ prepare a buffer for the next round
40: **end if**
41: **return** $winner.response[i]$
42: **end if**
43: **end for**

is increasing. At this moment, round numbers are assumed to be unbounded for the sake of simplicity and solutions to make round numbers bounded are presented in Section 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 37). 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 Algo. 6. These make the RMW procedure satisfy the requirement for using the LONGLIVEDCONSENSUS procedure (cf. Requirement 3.1).

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 4-5). The read value is checked to ensure that it is the result of round $(round_i - 1)$ (lines 6-14) (cf. Lemma 4.3). 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 Algo. 6). Only responses/toggle-bits corresponding to the processes that have submitted a new function to FUN, are updated to new values (lines 18-22). 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 25). 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 37) before returning the response $winner.response[i]$ (line 41).

Each process p_i has two buffers: 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 39).

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 address of a buffer containing a proposal. If the proposal contains a response 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 a process p_i writes its current round $round_i$ to $COU[i]$ (Algo. 7, line 37) only if $round_i$ has finished (i.e. an agreement among participating processes of $round_i$ has been achieved), a process p_n obtain a round number $round_n = \max_{1 \leq k \leq N} COU[k] + 1$ (Algo. 7, line 4) only if the round $round_n - 1$ has finished. □

Lemma 4.2. *The value res_k used from line 11 is a correct copy of $p_{res_k.owner}$'s shared buffer.*

Proof. The problem may happen is that when p_i makes a copy res_k of $WINNER[k]$ buffer (line 5), the buffer has been re-used (or has become the working buffer) for a later round. Note that $WINNER[k]$ contains the 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 5. Since i) $WINNER[k]$ and $COU[k]$ are updated in one atomic step using M_ASSIGNMENT (line 37) and ii) $COU[k]$ is always increasing, $cou_i[k] \leq round_a$.

Let p_o be the owner of $Buffer_1$. Since $Buffer_1$ is now p_o's currently *working* buffer in a round $round_b$, there exists a smallest round $round_e$, $round_a < round_e < round_b$, in which p_o was again the winner (line 39 is the only place p_o switches its *working* and *shared* buffers). Since $COU[k]$ is updated with $round_e$ (line 37) before the buffer $Buffer_1$ is switched from p_o's shared buffer to p_o's working buffer in order to be reused (line 39), $COU[k]$ was changed to $round_e$ before p_i finishes copying $Buffer_1$. Since p_o's round number is always increasing, $COU[o] \geq round_e > round_a \geq cou_i[k]$, which makes the algorithm either return earlier (line 7) or retry to read the value again (line 9), a contradiction to the hypothesis that this res_k value is used from line 11. □

Lemma 4.3. *The value res_k used to make p_i's proposal in round $round_i$ (line 17, Algo. 7) is the result of round $(round_i - 1)$.*

Proof. Due to Lemma 4.2, res_k from line 11 is the result of the latest round that p_k has finished until the time p_i reads that value at line 5. That round number is recorded in $res_k.round$. Since i) from line 17 $round_i > res_k.round$ (otherwise, the procedure returned at line 12 or retried at line 14) and ii) $res_k.round \geq cou_i[k]$(since the round number is always increasing) and iii) $cou_i[k] = (round_i - 1)$ (line 4), we have $round_i > res_k.round \geq (round_i - 1)$. Therefore, $res_k.round = (round_i - 1)$ or, in other words, res_k used at line 17 is the result of round $(round_i - 1)$. □

Lemma 4.4. *After a process p_i retries at line 9, 14, 32 or 35 in Algo. 7, 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 18. The winner of round $(round_i + 1)$ will realize that $FUN[i]$ has not been executed (line 20) since res_k is the result of round $round_i$ due to Lemma 4.3. 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 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 4.5. *Every process p_i will return with the response to its function after at most 2 iterations (line 3, Algo. 7).*

Proof. From Lemma 4.4, 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 7, 12 or 41, 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 25), making the procedure returns $\perp$ (line 26).

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 28-30). Therefore, p_i will return with the response to its function after executing at most 2 iterations. □

Lemma 4.6. *The* RMW *procedure is linearizable.*

Proof. (Sketch) Assume that RMW(f) 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 that has been executed in a round will never be executed in later rounds. Indeed, since i) a function and its toggle bit are atomically declared only once at the beginning of RMW by its unique owner/process (line 2) and ii) the value res_k used to make p_i's proposal in round $round_i$ (line 17) is the result of round $(round_i - 1)$ (Lemma 4.3) and iii) executing a function and updating its toggle bit in the result of a round by a process p_i occur atomically to other processes due to the way of constructing p_i's proposal (lines 21-22), p_i can check whether p_j's function has been executed by just comparing $FUN[j].toggle$ and $res_k.toggle[i]$ (line 20).

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 4.7. *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 3.4) and RMW returns after at most two iterations of its for-loop, the time complexity of RMW is $O(N)$. This also implies that RMW is wait-free. □

Lemma 4.8. *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. Algo. 6), we see that the $Proposal$ record has space complexity $O(N)$, leading to the space complexity of the BUF and $WINNER$ arrays is $O(N^2)$. Since the space complexity of the LONGLIVEDCONSENSUS procedure, which is used in the RMW procedure (line 25, Algo. 7), is also $O(N^2)$ (cf. Lemma 3.6), 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 Algo. 8). Due to Lemma 3.5, the space complexity of general wait-free RMW objects using only the M_ASSIGNMENT operation and read/write registers is at least $O(N^2)$. This means the space complexity $O(N^2)$ of the new wait-free RMW object (Algo. 7) 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 $agreement = \perp$ **then**
  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 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 [5].

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 higher level.

The representative process of p_i's D siblings at level l will participate in the wait-free LONGLIVEDCONSENSUS procedure with 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).

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. Algo. 9).

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**

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** $l = \mathcal{K} - 1$ to 2 **do**
4C: $winner \leftarrow$ LONGLIVEDCONSENSUS$^l(buf_i)$ // Achieve an agreement among p_i's D siblings at level l about who is their representative. Returning 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**

The process checks whether it successfully climbs up to level 1 by calling the CANDIDATE procedure (line 3R and Algo. 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 14R). In order to reduce the contention level on the shared variables COU and $WINNER$, RESILIENTRMW delays for a while between two consecutive reads using the backoff mechanism [3].

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_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 $O((\frac{N}{M})^2)$ [2]. This leads to the following lemma:

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

Proof. The time complexity of RMW using M_SCAN with time complexity $O((\frac{N}{M})^2)$ is $\max\{O((\frac{N}{M})^2), O(N)\}$. Since CANDIDATE invokes LONGLIVEDCONSENSUS with time complexity $O(D)$ at each of $\log N$ levels, the time complexity of CANDIDATE is $O((2M-2)\log N)$. Therefore, the time complexity of RESILIENTRMW is $\max\{O((\frac{N}{M})^2), O(N)\} + O((2M-2)\log N)$. If M is a constant, the time complexity becomes $O(N^2)$. If $\frac{N}{M} = \alpha$, where α is a constant, the the complexity becomes $O(N \log N)$. □

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

Proof. (Sketch) We will prove that correct processes always return with the response to its function if at most $(2M-3)$ RESILIENTRMW accesses to 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 processes submit a 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 Bounded round numbers

Active processes p_i that are participating in the most recent instance of the long-lived protocol need a mechanism to distinguish them from slow/sleepy processes. The bounded version of the long-lived protocol can be obtained by replacing the unbounded round number with the (bounded) *leadership graph* suggested in [9]. In the graph, an incoming process p_i invokes the ADVANCE operation to become one of the leaders of the graph. Processes that are current leaders belong to the most recent round whereas processes that are no longer leaders are slow processes. Therefore, the leadership graph can help distinguish active processes from slow processes, satisfying the requirement of the long-lived protocol. Another approach to bound the round number is to use the transforming technique presented in [7]. The technique can transform any unbounded algorithm based on an asynchronous rounds structure into a bounded algorithm in a way that preserves correctness and running time.

[4] *Correct* processes are processes that do not crash in the object execution.

7 Conclusions

In this paper, based on the intrinsic features of emerging media/graphics processing unit architectures we have generalized the architectures to an abstract model of a chip with multiple SIMD cores sharing a memory. For this general model, which does not support strong synchronization primitives like *test-and-set* and *compare-and-swap*, we have developed a wait-free *long-lived* consensus object for $N = (2M - 2)$ cores, where M is 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 [10]. 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 $O(N)$. 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 synchronization mechanisms in recent GPU architectures and the need of synchronization mechanisms in parallel applications. The results show that wait-free programming is possible for GPUs, extending the set of parallel applications that can utilize the ubiquitous and powerful computational hardware. Last but not least, the results demonstrate that it is possible to construct wait-free synchronization mechanisms for GPUs without the need of strong synchronization primitives in hardware, implying more transistors in GPUs can be devoted to data processing and intensive computing instead of strong synchronization primitives.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.0.* NVIDIA Corporation, 2007.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Procs. of the Annual Intl. Symp. on Computer Architecture*, pages 396–406, 1989.

[4] 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*, pages 91–100, 1993.

[5] 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*, 34(2):333–357, 2005.

[6] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[7] C. Dwork and M. Herlihy. Bounded round number. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 53–64, 1993.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[9] M. Herlihy. Randomized wait-free concurrent objects (extended abstract). In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 11–21, 1991.

[10] M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, 1991.

[11] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[12] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[13] S. S. Lumetta and D. E. Culler. Managing concurrent access for shared memory active messages. In *Proc. of the Intl. Parallel Processing Symp. (IPPS)*, page 272, 1998.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[15] M. M. Michael and M. L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proc. of the IEEE Intl. Parallel Processing Symp. (IPPS*, pages 267–273, 1997.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[17] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[18] 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*, pages 184–185, 2005.

[19] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 320–321, 2001.

[20] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proc. of the ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67, 2002.