NB-FEB: A Universal Scalable Easy-to-Use Synchronization Primitive for Manycore Architectures

Phuong Hoai Ha[1], Philippas Tsigas[2], and Otto J. Anshus[1]

[1] University of Tromsø, Department of Computer Science, Faculty of Science, NO-9037 Tromsø, Norway, {phuong,otto}@cs.uit.no
[2] Chalmers University of Technology, Department of Computer Science and Engineering, SE-412 96 Göteborg, Sweden, tsigas@chalmers.se

Abstract. This paper addresses the problem of universal synchronization primitives that can support *scalable* thread synchronization for large-scale manycore architectures. The universal synchronization primitives that have been deployed widely in conventional architectures, are the *compare-and-swap* (CAS) and *load-linked/store-conditional* (LL/SC) primitives. However, such synchronization primitives are expected to reach their scalability limits in the evolution to manycore architectures with thousands of cores.

We introduce a *non-blocking* full/empty bit primitive, or NB-FEB for short, as a promising synchronization primitive for parallel programming on manycore architectures. We show that the NB-FEB primitive is *universal, scalable, feasible* and *easy to use*. NB-FEB, together with registers, can solve the consensus problem for an arbitrary number of processes (*universality*). NB-FEB is *combinable*, namely its memory requests to the same memory location can be combined into only one memory request, which consequently makes NB-FEB scalable (*scalability*). Since NB-FEB is a variant of the original full/empty bit that always returns a value instead of waiting for a conditional flag, it is as feasible as the original full/empty bit, which has been implemented in many computer systems (*feasibility*). We construct, on top of NB-FEB, a non-blocking software transactional memory system called NBFEB-STM, which can be used as an abstraction to handle concurrent threads *easily*. NBFEB-STM is space efficient: the space complexity of each object updated by N concurrent threads/transactions is $\Theta(N)$, which is optimal.

1 Introduction

Universal synchronization primitives [9] are essential for constructing non-blocking synchronization mechanisms for parallel programming, such as non-blocking software transactional memory [8, 10, 12, 17]. Non-blocking synchronization eliminates the concurrency control problems of mutual exclusion locks, such as priority inversion, deadlock and convoying. As manycore architectures with thousands of cores are expected to be our future chip architectures [2], universal synchronization primitives that can support scalable thread synchronization for such large-scale architectures are desired.

However, the conventional universal primitives such as *compare-and-swap* (CAS) and *load-linked/ store-conditional* (LL/SC) are expected to reach their scalability limits in the evolution to manycore architectures with thousands of cores. The universal primitives are usually built on top of conventional cache-coherent protocols. Experimental studies have recently shown that the universal primitives, which lock the entire memory bank to ensure atomicity (i.e. coarse-grained synchronization), are not scalable for multicore architectures [19]. The authors of [19] also experimentally show that the original (blocking) *full/empty bit* (FEB), which lock only the memory location under consideration (i.e. fine-grained synchronization), scales much better. Moreover, the conventional cache-coherent protocols are considered inefficient for large-scale manycore architectures [2]. As a result, several emerging multicore architectures, such as the NVIDIA CUDA, the ClearSpeed CSX, the IBM Cell BE and the Cyclops-64, utilize a fast local memory for each processing core rather than a coherent data cache.

For the emerging manycore architectures without a coherent data cache, the CAS and LL/SC primitives are not scalable either since they are not *combinable* [3, 11]. Primitives are combinable if their memory requests to the same memory location (arriving at a switch of the processor-to-memory interconnection network) can be combined into *only one* memory request. Separate replies to the original requests are later created from the reply to the combined request (at the switch). The combining technique has been implemented in the NYU Ultracomputer [4] and the IBM RP3 machine [14], and has been shown to be a promising technique for large-scale multiprocessors to alleviate the performance degradation due to a synchronization "hot spot". The CAS primitives are not combinable since the success of a $CAS(x, a, b)$ primitive depends on the current value of the memory location x. For m-bit locations (e.g. 64-bit words), there are 2^m possible values and therefore, a combined request that represents k $CAS(x, a, b)$ requests, $k < 2^m$, must carry as many as k different checking-values a and k new values b. Although the *single-valued* $CAS_a(x, b)$ [3], which will atomically swap b to x if x equals a is combinable, the number of instructions CAS_a must be as many as the number of integers a that can be stored in one memory word (e.g. 2^{64} CAS_a instructions for 64-bit words, where $a = 0, 1 \cdots, 2^{64} - 1$). Note that the value domains of x, a and b must be the same. This fact makes the *single-valued* CAS_a unfeasible for hardware implementation. Note that the LL/SC primitives are not combinable since the success of a SC primitive depends on the state of its reservation bit at the memory location that has been set previously by the corresponding LL primitive. Therefore, a combined request that represents k SC requests (from different processes/processors) must carry as many as k store values.

Another universal primitive called *sticky bit* has been suggested in [15], but it has not been deployed so far due to its usage complexity. A sticky bit is a data object that holds 0, 1 or $\perp$ and supports the following operations: $Jam(v)$, which sets the value to v and returns $Success$ atomically if the value was $\perp$ or v; $Flush()$, which sets the value to $\perp$; and $Read()$, which returns the current value of the object. To the best of our knowledge, the universal construction using the sticky bit in [15] does not prevent a delayed thread, even after being helped, from jamming the sticky bits of a cell that has been re-initialized and reused. Since the universal construction is built on a doubly-linked list of cells, it is not obvious how an external garbage collector (supported by the underlying system) can help solve the problem. Moreover, the space complexity of the universal construction for an object is as high as $O(N^3)$ [15] [3] , where N is the number of processes.

This paper suggests a novel synchronization primitive, called NB-FEB, as a promising synchronization primitive for parallel programming on manycore architectures. What makes NB-FEB a promising primitive is its following four main properties. NB-FEB is:

Feasible: NB-FEB is a *non-blocking* variant of the conventional full/empty bit that *always returns* the old value of the variable instead of waiting for its conditional flag to be set (or cleared) (cf. Section 3). This simple modification makes NB-FEB as *feasible* as the original (blocking) full/empty bit, which has been implemented in many computer systems such as HEP, Tera (or Cray MTA-2), MDP, Sparcle, M-Machine, and Eldorado. The original full/empty bit is also used to design a *synchronization array* – a dedicated hardware structure for pipelined inter-thread communication [16].

Universal: This simple modification, however, significantly increases the synchronization power of full/empty bits, making NB-FEB as powerful as CAS or LL/SC. NB-FEB, together with registers, can solve wait-free[4] consensus [9] for an arbitrary number of processes, the essential property for constructing non-blocking synchronization mechanisms (cf. Section 3.1). Note that due to *blocking*, the original full/empty bit is as weak as read/write registers with respect to synchronization power: it, together with registers, cannot solve *wait-free* consensus for even two processes.

Scalable: NB-FEB is *combinable*, namely its memory requests to the same memory location can be combined into only one memory request (cf. Section 3.2). This empowers NB-FEB with the ability to provide *scalable* thread synchronization for large-scale manycore architectures [4, 14].

Easy-to-use: The original full/empty bit is well-known as a *special-purpose* primitive for fast producer-consumer synchronization and has been used extensively in specific domains of applications (e.g. parallel graph algorithms). In this paper, by providing an abstraction on top of NB-FEB, we show that NB-FEB can be deployed easily as a *general-purpose* primitive. We construct, on

[3] Each object needs $O(N^2)$ cells of size $O(N)$.

[4] An implementation is *wait-free* if it guarantees that any process can complete any operation on the implemented object in a finite number of steps, regardless of the execution speeds on the other processes.

top of NB-FEB, a non-blocking software transactional memory system called NBFEB-STM, which can be used as an abstraction to handle concurrent threads *easily*. NBFEB-STM is space efficient: the space complexity of each object updated by N concurrent threads/transactions is $\Theta(N)$, the optimal (cf. Section 4).

The rest of this paper is organized as follows. Section 2 presents the shared memory and interconnection network models assumed in this paper. Sections 3 describes the NB-FEB primitive in detail and proves its universality and combinability properties. Section 4 introduces and analyzes NBFEB-STM, the obstruction-free multi-versioning STM constructed on top of the NB-FEB primitive. Section 5 concludes this paper. Because of space limitations, most proofs are omitted from this version of the paper and can be found in [7].

2 Models

Similarly to previous research on the synchronization power of synchronization primitives [9], this paper assumes the linearizable shared memory model. Due to NB-FEB combinability, as in [11] we assume that the processor-to-memory interconnection network is *nonovertaking* and that a reply message is sent back on the same path followed by the request message. The intermediate nodes on the communication path from a processor to a global shared memory module can be either switches of a multistage interconnection network [11] or memory modules of a multilevel memory hierarchy [3]. The intermediate nodes can detect requests destined for the same destination and maintain the queues of requests. In this paper, we assume that such a combining network is provided and we mainly focus on the combining logic of the new primitive. For the design details of the combining network, the reader is referred to [4]. No memory coherent schemes are assumed.

3 NB-FEB Primitives

NB-FEB is a set of four primitives: *test-flag-and-set TFAS* (cf. Algorithm 1), *Load* (Algorithm 2), *store-and-clear SAC* (Algorithm 3) and *store-and-set SAS* (Algorithm 4). Each variable x has an associated full-empty bit $flag_x$. Primitive *TFAS* will atomically write value v to variable x (and set $flag_x$ to $true$) if $flag_x$ is $false$. The primitive always returns the (previous) value of pair $(x, flag_x)$ regardless of the value of $flag_x$. Primitive SAC atomically writes v to x, sets $flag_x$ to $false$ and returns the previous value of $(x, flag_x)$. Primitive SAS is similar to SAC except that SAS sets $flag_x$ to $true$. Regarding conditional load primitives such as *load-if-set* and *load-if-clear* in the original FEB, a processor can check the flag value, $flag_x$, returned by the unconditional load primitive *Load* to determine if it was successful.

When the value of $flag_x$ returned is not needed, we just write $r \leftarrow \text{TFAS}(x, v)$ instead of $(r, flag_r) \leftarrow \text{TFAS}(x, v)$, where r is x's old value. The same applies to

SAC and *SAS*. For *Load*, we just write $r \leftarrow x$ instead of $r \leftarrow$ LOAD(x). In this paper, the flag value returned is needed only for combining NB-FEB primitives.

Algorithm 1 TFAS(x: variable, v: value): Test-Flag-And-Set, a non-blocking variant of the original Store-if-Clear-and-Set primitive, which *always* returns the old value of x.

```
$(o, flag_o) \leftarrow (x, flag_x)$;
if $flag_x =$ false then
    $(x, flag_x) \leftarrow (v,$ true$)$;
end if
return $(o, flag_o)$;
```

Algorithm 2 LOAD(x: variable)

```
return $(x, flag_x)$;
```

Algorithm 3 SAC(x: variable, v: value): Store-And-Clear

```
$(o, flag_o) \leftarrow (x, flag_x)$;
$(x, flag_x) \leftarrow (v,$ false$)$;
return $(o, flag_o)$;
```

Algorithm 4 SAS(x: variable, v: value): Store-And-Set

```
$(o, flag_o) \leftarrow (x, flag_x)$;
$(x, flag_x) \leftarrow (v,$ true$)$;
return $(o, flag_o)$;
```

Algorithm 5 SICAS(x: variable, v: value): Store-If-Clear-And-Set, one of the original *FEB* primitives, which waits for $flag_x$ to be clear (or *false*).

```
Wait for $flag_x$ to be false;
$(x, flag_x) \leftarrow (v,$ true$)$;
```

Algorithm 6 LISAC(x: variable): Load-If-Set-And-Clear, one of the original *FEB* primitives, which waits for $flag_x$ to be set (or *true*).

```
Wait for $flag_x$ to be true;
$flag_x \leftarrow$ false;
return $x$;
```

Algorithm 7 TFAS_CONSENSUS(*proposal*: value)

Decision: shared variable. The shared variable is initialized to $\perp$ with a clear flag (i.e. $flag_{Decision} =$ **false**).

```
Output: a value agreed by all processes.
1T: $(first, flag_{first}) \leftarrow$ TFAS$(Decision, proposal)$;
2T: if $first = \perp$ then
3T:     return $proposal$;
4T: else
5T:     return $first$;
6T: end if
```

3.1 *TFAS*: A Universal Primitive

In this section, we will show that *TFAS* is a universal primitive like *CAS*. Note that due to blocking, the original full/empty bit primitives such as *store-if-clear-and-set* (cf. Algorithm 5) and *load-if-set-and-clear* (cf. Algorithm 6) are as weak as read/write registers with respect to synchronization power: they, together with registers, cannot solve *wait-free* consensus [9] for even two processes.

Lemma 1. *(Universality) The* test-flag-and-set *primitive (or TFAS for short) is universal.*

The wait-free consensus algorithm is shown in Algorithm 7. Processes share a variable called *Decision*, which is initialized to $\perp$ with a *false* flag. Each process *p* proposes its value ($\neq \perp$) called *proposal* by calling TFAS_CONSENSUS (*proposal*).

The TFAS_CONSENSUS procedure is clearly wait-free since it contains no loops. It is not difficult to see that the procedure will return the proposal of the first process executing *TFAS* on the *Decision* variable to all processes.

3.2 Combinability

Lemma 2. *(Combinability) NB-FEB primitives are combinable.*

Proof. Figure 1 summarizes the combining logic of NB-FEB primitives on a memory location x. The first column is the name of the first primitive request and the first row is the name of the successive primitive request. For instance, the cell [*SAS*, *TFAS*] is the combining logic of *SAS* and *TFAS* in which *SAS* is followed by *TFAS*. Let v_1, v_2, r and f_r be the value of the first primitive request, the value of the second primitive request, the value returned and the flag returned, respectively. In each cell, the first line is the combined request, the second is the reply to the first primitive request and the third (and forth) is the reply to the successive primitive request. The values 0 and 1 of f_r in the reply represent $false$ and $true$, respectively.

Consider cell [*TFAS*, *TFAS*] as an example. The cell describes the case where request $TFAS(x, v_1)$ is followed by request $TFAS(x, v_2)$, at an intermediate node (e.g. a switch) of the processor-to-memory interconnection network. At the node, the two input requests are combined into only one output request $TFAS(x, v_1)$ (line 1), which will be forwarded further to the corresponding memory controller. When receiving a reply (r, f_r) to the combined request, the intermediate node at which the requests were combined, creates separate replies to the two original requests. The reply to the first original request, $TFAS(x, v_1)$, is (r, f_r) (line 2) as if the request was executed by the memory controller. The reply to the successive request, $TFAS(x, v_2)$, depends on whether the combined request $TFAS(x, v_1)$ has successfully updated the memory location x. If $f_r = 0$, $TFAS(x, v_1)$ has successfully updated x with its value v_1. Therefore, the reply to the successive request $TFAS(x, v_2)$ is $(v_1, 1)$ as if the request was executed right after the first request $TFAS(x, v_1)$. If $f_r = 1$, $TFAS(x, v_1)$ has failed to update the x variable. Therefore, the reply to the successive request $TFAS(x, v_2)$ is $(r, 1)$. □

Due to the combining logic in Figure 1, the set of primitives *TFAS* (universal primitive), *Load* (read-primitive), *SAC* and *SAS* (write-primitives) are closed under the combining operation: the combination of any two primitives of the set belongs to the set (e.g. cell [*SAC*, *TFAS*] is *SAS*). Namely, all concurrent requests to the same memory location can be combined into *only one* request. Based on previous experimental results of combinable primitives in the literature such as *fetch-and-add* [4, 14], NB-FEB would be scalable in practice.

4 NBFEB-STM: Obstruction-free Multi-versioning STM

Like previous obstruction-free [5] multi-versioning STM called LSA-STM [17], the new software transactional memory called NBFEB-STM assumes that objects are only accessed and modified within transactions. NBFEB-STM assumes that

[5] A synchronization mechanism is *obstruction-free* if any thread that runs solo (i.e. without encountering a synchronization conflict from a concurrent thread) for long enough, makes progress.

there are no nested transactions, namely each thread executes only one transaction at a time. NBFEB-STM, like other obstruction-free STMs [10, 12, 17], is designed for garbage-collected programming languages (e.g. Java). A variable reclaimed by the garbage collector is assumed to have all bits 0 when it is reused. Note that there are non-blocking garbage collection algorithms that do not require synchronization primitives other than reads and writes while they still guarantee the non-blocking property for application-threads. Such a garbage collection algorithm is presented in [7].

$(x, [v_1])$	The successive primitive with parameters $(x, [v_2])$			
	Load	*SAC*	*SAS*	*TFAS*
Load	*Load* (r, f_r) (r, f_r)	$SAC(v_2)$ (r, f_r) (r, f_r)	$SAS(v_2)$ (r, f_r) (r, f_r)	$TFAS(v_2)$ (r, f_r) (r, f_r)
SAC	$SAC(v_1)$ (r, f_r) $(v_1, 0)$	$SAC(v_2)$ (r, f_r) $(v_1, 0)$	$SAS(v_2)$ (r, f_r) $(v_1, 0)$	$SAS(v_2)$ (r, f_r) $(v_1, 0)$
SAS	$SAS(v_1)$ (r, f_r) $(v_1, 1)$	$SAC(v_2)$ (r, f_r) $(v_1, 1)$	$SAS(v_2)$ (r, f_r) $(v_1, 1)$	$SAS(v_1)$ (r, f_r) $(v_1, 1)$
TFAS	$TFAS(v_1)$ (r, f_r) Like 5th column	$SAC(v_2)$ (r, f_r) Like 5th column	$SAS(v_2)$ (r, f_r) Like 5th column	$TFAS(v_1)$ (r, f_r) if f_r=0: $(v_1, 1)$ else: $(r, 1)$

Fig. 1. The combining logic of NB-FEB primitives on a memory location x

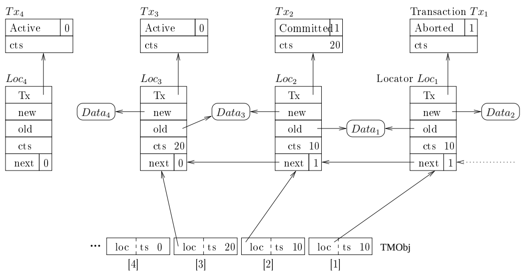

Fig. 2. The data structure of a transactional memory object *TMObj* in NBFEB-STM.

Primitives *TFAS*, *SAC* and *Load* are used to implement NBFEB-STM. Note that primitive *SAS* is included in NB-FEB to make the set of NB-FEB primitives closed under the combining operation (cf. cell [*SAC*, *TFAS*] in Figure 2). Since NB-FEB primitives are combinable, NBFEB-STM eliminates all conventional synchronization hot spots in STMs (cf. Lemma 10).

4.1 Challenges and Key Ideas

Unlike the STMs using *CAS* [10, 12, 17], NBFEB-STM using *TFAS* and *SAC* must handle the problem that *SAC*'s interference with concurrent *TFAS* will violate the atomicity semantics expected on variable x. Overlapping $TFAS_1$ and $TFAS_2$ both may successfully write their new values to x if *SAC* interference occurs.

The key idea is not to use the transactional memory object *TMObj* [6] [10, 12, 17] that needs to switch its pointer frequently to a new locator (when a transaction commits). Such a *TMObj* would need *SAC* in order to clear the pointer's flag, allowing the next transaction to switch the pointer. Instead, NBFEB-STM keeps a linked-list of locators for each object and integrates a write-once pointer *next* into each locator (cf. Figure2). When opening an object O for write, a

[6] The reader is assumed to be familiar with the basic concepts of STMs, which are nicely presented in the seminal paper [10].

transaction T tries to append its locator to O's locator-list by changing the *next* pointer of the head-locator[7] of the list using TFAS. Due to the semantics of TFAS, only one of the concurrent transactions trying to append their locators succeeds. The other transactions must retry in order to find the new head and then append their locators to the new head. Using the locator-list, each *next* pointer is changed only once and thus its flag does not need to be cleared during the lifetime of the corresponding locator. This prevents SAC from interleaving with concurrent TFAS. The *next* pointer, together with its locator, will be reclaimed by the garbage collector when the lifetime of the locator is over. The garbage collector ensures that a locator will not be recycled until no thread/transaction has a reference to it.

Linking locators together creates another challenge on the space complexity of NBFEB-STM. Unlike the STMs using CAS, a delayed/halted transaction T in NBFEB-STM may prevent all locators appended after its locator in a locator-list from being reclaimed. As a result, T may make the system run out of memory and thus prevent other transactions from making progress, violating the obstruction-freedom property. The key idea to solve the space challenge is to break the list of obsolete locators into pieces so that a delayed transaction T prevents from being reclaimed only the locator that T has a direct reference as in the STMs using CAS. The idea is based on the fact that only the head of O's locator-list is needed for further accesses to the O object.

However, breaking the list of an obsolete object O also creates another challenge on finding the head of O's locator-list. Obviously, we cannot use a head pointer as in non-blocking linked-lists since modifying such a pointer requires CAS. The key idea is to utilize the fact that there are no nested transactions and thus each thread has at most one *active* locator[8] in each locator list. Therefore, by recording the latest locator of each thread appended to O's locator-list, a transaction can find the head of O's locator list. The solution is elaborated further in Section 4.3 and Section 4.4.

Based on the key ideas, we come up with the data structure for a transactional memory object that is illustrated in Figure 2 and presented in Algorithm 8.

4.2 Data Structures

A transactional memory object TMObj in NBFEB-STM is an array of N pairs (pointer, timestamp), where N is the number of concurrent threads/transactions, as shown in Figure 2. Item $\mathit{TMObj}[i]$ is modified only by thread t_i and can be read by all threads. Pointer $\mathit{TMObj}[i].\mathit{loc}$ points to the locator called Loc_i corresponding to the latest transaction committed/aborted by thread t_i. Timestamp $\mathit{TMObj}[i].\mathit{ts}$ is the commit timestamp of the object referenced by $\mathit{Loc}_i.\mathit{old}$. After successfully appending its locator Loc_i to the list by executing $\mathit{TFAS}(\mathit{head.next}, \mathit{Loc}_i)$, t_i will update its own item $\mathit{TMObj}[i]$ with its new locator Loc_i. The TMObj array is used to find the head of the list of locators $\mathit{Loc}_1, \cdots, \mathit{Loc}_N$. Note that since

[7] The head-locator is the last element of O's singly-linked list whose next pointer is *null* (e.g. Loc_3 in Figure 2)

[8] An *active* locator is a locator that is still in use, opposite to an *obsolete* locator.

contemporary large-scale multicore architectures (e.g. NVIDIA CUDA with up to 30 cores [13]) deploy high memory bandwidth to deal with the high latency of shared memory accesses, reading/writing to an array with several elements such as *TMObj* will not be problematic on manycore architectures. For instance, the NVIDIA CUDA currently allows each core to read/write to a large segment of shared memory (e.g. 128 bytes) in a single memory transaction.

For each locator Loc_i, in addition to fields Tx, *old* and *new* that reference the corresponding transaction object, the old data object and the new data object, respectively, as in DSTM[10], there are two other fields *cts* and *next*. The *cts* field records the commit timestamp of the object referenced by *old*. The *next* field is the pointer to the next locator in the locator list. The *next* pointer is modified by NB-FEB primitives.

In Figure 2, values $\{0, 1\}$ in the *next* pointer denote the values $\{false, true\}$ of its flag, respectively. The *next* pointer of the head of the locator list, $Loc_3.next$, has its flag clear (i.e. 0) while the *next* pointers of previous locators (e.g. $Loc_1.next$, $Loc_2.next$) have their flags set (i.e. 1) since their *next* pointers were changed. The *next* pointer of a new locator (e.g. $Loc_4.next$) is initialized to $(\perp, 0)$. Due to the garbage collector semantics, all locators Loc_j reachable from the *TMObj* shared object by following their $Loc_j.next$ pointers, will not be reclaimed.

For each transaction object Tx_i, in addition to fields *status*, *readSet* and *writeSet* corresponding to the status, the set of objects opened for read, and the set of objects opened for write, respectively, there is a field *cts* recording Tx_i's commit timestamp (if Tx_i committed) as in LSA-STM [17].

4.3 Algorithm

A thread t_i starts a transaction T by calling the StartSTM(T) procedure (Algorithm 8). The procedure sets $T.status$ to *Active* and clears its flag using *SAC* (cf. Algorithm 3). The procedure then initializes the lazy snapshot algorithm (LSA) [17] by calling LSA_Start. NBFEB-STM utilizes LSA to preclude inconsistent views by *live* transactions, an essential aspect of transactional memory semantics [6]. The LSA has been shown to be an efficient mechanism to construct consistent snapshots for transactions [17]. Moreover, the LSA can utilize up to $(N + 1)$ versions of a transactional memory object *TMObj* recorded in N locators of *TMObj*'s locator list. Note that the global counter *CT* in LSA can be implemented by the *fetch-and-increment* primitive [4], a combinable (and thus scalable) primitive [11]. Except for the global counter *CT*, the LSA in NBFEB-STM does not need any strong synchronization primitives other than *TFAS*. The Abort(T) operation in LSA, which is used to abort a transaction T, is replaced by *TFAS*($T.status$, *Aborted*). Note that the *status* field is the only field of a transaction object T that can be modified by other transactions.

Read-accesses When a transaction T opens an object O for read, it invokes the OpenR procedure (Algorithm 11). The procedure simply calls the LSA_Open procedure of LSA [17] in the *Read* mode to get the version of O that maintains a consistent snapshot with the versions of other objects being accessed by T.

If no such a version of O exists, LSA_OPEN will abort T and consequently OPENR will return $\perp$ (line 3R). That means there is a conflicting transaction that makes T unable to maintain a consistent view of all the object being accessed by T. Otherwise, OPENR returns the version of O that is selected by LSA. This version is guaranteed by LSA to belong to a consistent view of all the objects being accessed by T. Up to $(N + 1)$ versions are available for each object O in NBFEB-STM (cf. Lemma 7). Since NBFEB-STM utilizes LSA, read-accesses to an object O are invisible to other transactions and thus do not change O's locator list.

Write-accesses When a transaction T opens an object O for write, it invokes the OPENW procedure (cf. Algorithm 13). The task of the procedure is to append to the head of O's locator list a new locator L whose Tx and *old* fields reference to T and O's latest version, respectively. In order to find O's latest version, the procedure invokes FINDHEAD (cf. Algorithm 10) to find the current head of O's locator list (line 3W). When the head called H is found, the procedure determines O's latest version based on the status of the corresponding transaction $H.Tx$ as in DSTM [10], by invoking INITLOC (line 4W). If the $H.Tx$ transaction committed, O's latest version is $H.new$ with commit timestamp $H.Tx.cts$ (lines 2I-4I, Algorithm 12). A copy of O's latest version is created and referenced by $L.new$ (line 5I) (cf. locators Loc_2 and Loc_3 in Figure 2 as H and L, respectively, for an illustration). If the $H.Tx$ transaction aborted, O's latest version is $H.old$ with commit timestamp $H.cts$ (lines 7I-9I) (cf. locators Loc_1 and Loc_2 in Figure 2 as H and L, respectively, for an illustration). If the $H.Tx$ transaction is active, OPENW consults the contention manager [5, 18] (line 13I, Algorithm 12) to solve the conflict between the T and $H.Tx$ transactions. If T must abort, OPENW tries to change $T.status$ to *Aborted* using *TFAS* (lines 15I-16I, Algorithm 12) and returns $\perp$ (line 5W, Algorithm 13). Note that other transactions change $T.status$ only to *Aborted*, and thus if *TFAS* at line 15I fails, $T.status$ has been changed to *Aborted* by another transaction. If $H.Tx$ must abort, OPENW changes $H.Tx.status$ to *Aborted* using *TFAS* (line 18I, Algorithm 12) and checks $H.Tx.status$ again.

The latest version of O is then checked to ensure that it, together with the versions of other objects being accessed by T, belongs to a consistent view using LSA_OPEN with "Write" mode (line 7W). If it does, OPENW tries to append the new locator L to O's locator list by changing the $H.next$ pointer to L (line 11W). Note that the $H.next$ pointer was initialized to $\perp$ with a clear flag, before H was successfully appended to O's locator list (line 24I, Algorithm 12). If OPENW does not succeed, another locator has been appended as a new head and thus OPENW must retry to find the new head (line 12W). Otherwise, it successfully appends the new locator L as the new head of O's locator list. OPENW, which is being executed by a thread t_i, then makes $O[i].ptr$ reference to L and records $L.cts$ in $O[i].ts$ (line 15W). This removes O's reference to the previous locator *oldLoc* appended by t_i, allowing *oldLoc* to be reclaimed by the garbage collector. Since *oldLoc* now becomes an obsolete locator, its *next* pointer is reset (line 16W) to break possible chains of obsolete locators reachable by a delayed/halted thread, helping *oldLoc*'s descendant locators in the chains be reclaimed. For each item j in the O array such that timestamp $O[j].ts < O[i].ts$, the $O[j].ptr$ locator now becomes obsolete in a sense that it no longer keeps O's latest version although it is still referenced by $O[j]$ (since only thread t_j can modify $O[j]$). In order to break the chains of obsolete locators, OpenW resets the *next* pointer of the $O[j].ptr$ locator so that $O[j].ptr$'s descendant locators can be reclaimed by the garbage collector (lines 17W-18W). This chain-breaking mechanism makes the space complexity of an object updated by N concurrent transactions/threads in NBFEB-STM be $\Theta(N)$, the optimal (cf. Theorem 1).

In order to find the head of O's locator list as in OpenW, a transaction invokes the FindHead(O) procedure (cf. Algorithm 10). The procedure atomically reads O into a local array *start* (line 2F). Such a multi-word read operation is supported by emerging multicore architectures (e.g. CUDA [13]) which deploy high memory bandwidth to deal with the high latency of shared memory accesses. In the contemporary chips of these architectures, a read operation can atomically read 128 bytes. In general, such a multi-word read operation can be implemented as an atomic snapshot using only single-word read and single-word write primitives[9] [1]. FindHead finds the item $start_{latest}$ with the highest timestamp in *start* and searches for the head from locator $start_{latest}.ptr$ by following the *next* pointers until it finds a locator H whose *next* pointer is $\perp$ (lines 3F-6F). Since some locators may become obsolete and their *next* pointers were reset to $\perp$ by concurrent transactions (lines 16W and 18W in Algorithm 13), FindHead needs to check H's commit timestamp against the highest timestamp of O at a moment after H is found (lines 8F-10F). If H's commit timestamp is greater than or equal to the highest timestamp of O, H is the head of O's locator list (cf. Lemma 4). Otherwise, H is an obsolete locator and FindHead must retry (line 10F). The FindHead procedure is lock-free, namely it will certainly return the head of O's locator list after at most N iterations unless a concurrent thread has completed a transaction and subsequently has started a new one, where N is the number of concurrent (updating) threads (cf. Lemma 5). Note that as soon as a thread obtains *head* from FindHead (line 3W of OpenW, Algorithm 13), the locator referenced by *head* will not be reclaimed by the garbage collector until the thread returns from the OpenW procedure.

Commitments When committing, read-only transactions in NBFEB-STM do nothing and always succeed in their commit phase as in LSA-STM [17]. They can abort only when trying to open an object for read (cf. Algorithm 11). Other transactions T, which have opened at least one object for write, invoke the CommitW procedure (Algorithm 9). The procedure calls the LSA_Commit procedure to ensure that T still maintains a consistent view of objects being accessed by T (line 1C). T's commit timestamp is updated with the timestamp returned from LSA_Commit (line 2C). Finally, CommitW tries to change *T.status* to *Committed* (line 3C). *T.status* will be changed to *Committed* at this step if it has not been changed to *Aborted* due to the semantics of *TFAS*.

[9] Note that single-word read/write primitives are combinable [11].

Algorithm 8 StartSTM(T: transaction)

TMObj: array$[N]$ of $\{ptr, ts\}$. Pointer $TMObj[i].ptr$ points to the locator called Loc_i corresponding to the latest transaction committed/aborted by thread t_i. Timestamp $TMObj[i].ts$ is the commit timestamp of the object referenced by $Loc_i.old$. N is the number of concurrent threads/transactions. $TMObj[i]$ is written only by thread t_i.

Locator: **record** tx, new, old: pointer; cts: timestamp; **end**. The cts timestamp is the commit timestamp of the old version.

Transaction: **record** $status$: $\{Active, Committed, Aborted\}$; cts: timestamp; **end**. NBFEB-STM also keeps read/write sets as in LSA-STM, but the sets are omitted from the pseudocode since managing the sets in NBFEB-STM is similar to LSA-STM.

```
1S: SAC($T.status, Active$); // Store-and-clear
2S: LSA_Start($T$) // Lazy snapshot algorithm
```

Algorithm 9 CommitW(T: Transaction): Try to commit an update transaction T by thread p_i

```
1C: $CT_T \leftarrow$ LSA_Commit($T$); // Check consistent snapshot. $CT_T$ is $T$'s unique commit timestamp from LSA.
2C: $T.cts \leftarrow CT_T$; // Commit timestamp of $T$ if $T$ manages to commit.
3C: TFAS($T.status, Committed$);
```

Algorithm 10 FindHead(O: TMObj): Find the head of the locator list

Output: reference to the head of the locator list

```
1F: repeat
2F:     $start \leftarrow O$; // Read $O$ to a local array atomically.
3F:     Let $start_{latest}$ be the item with highest timestamp;
4F:     $tmp \leftarrow start_{latest}.ptr$; // Find a (possible) head, starting from $start_{latest}.ptr$.
5F:     while $tmp.next \neq \perp$ do
6F:         $tmp \leftarrow tmp.next$;
7F:     end while
8F:     $start' \leftarrow O$; // Check if current $tmp$ is the actual head.
9F:     Let $start'_{latest}$ be the item with highest timestamp;
10F: until $tmp.cts \geq start'_{latest}.ts$;
11F: return $tmp$;
```

Algorithm 11 OpenR(T: Transaction; O_i: TMObj): Open a transactional object for read

Output: reference to a *data* object if succeeds, or $\perp$.

```
1R: LSA_Open($T, 0_i$, "$Read$"); // LSA's Open procedure
2R: if $T.status = Aborted$ then
3R:     return $\perp$;
4R: else
5R:     return the version chosen by LSA_Open;
6R: end if
```

4.4 Analysis

In this section, we prove that NBFEB-STM fulfills the three essential aspects of transactional memory semantics [6]:

Instantaneous commit: Committed transactions must appear as if they executed instantaneously at some unique point in time, and aborted transactions, as if they did not execute at all.

Preluding inconsistent views: The state (of shared objects) accessed by *live* transactions must be consistent.

Preserving real-time order: If a transaction T_i commits before a transaction T_j starts, then T_i must appear as if it executed before T_j. Particularly, if a transaction T_1 modifies an object O and commits, and then another transaction T_2 starts and reads O, then T_2 must read the value written by T_1 and not an older value.

We present some key properties of NBFEB-STM that make NBFEB-STM fulfill the three aspects. Because of space limitations, proofs are in the full version of this paper [7].

Algorithm 12 INITLOC(*newLoc*, *head*: Locator; *T*: Transaction): Initialize a new locator

```
Output: $\perp$ if $T.status = Aborted$
1I: for $i = 0$ to $1$ do
2I:   if $head.tx.status = Committed$ then
3I:     $newLoc.old \leftarrow head.new$;
4I:     $newLoc.cts \leftarrow head.tx.cts$;
5I:     $newLoc.new \leftarrow$ COPY($head.new$);// Create a duplicate
6I:     break;
7I:   else if $head.tx.status = Aborted$ then
8I:     $newLoc.old \leftarrow head.old$;
9I:     $newLoc.cts \leftarrow head.cts$;
10I:    $newLoc.new \leftarrow$ COPY($head.old$);
11I:    break;
12I:  else
13I:    $myProgession \leftarrow$ CM($O_i$, "Write"); // head.tx is active $\Rightarrow$ Consult the contention manager
14I:    if $myProgression =$ false then
15I:      TFAS($T.status$, $Aborted$); // If fails, another has executed this TFAS.
16I:      return $\perp$;
17I:    else
18I:      TFAS($head.tx.status$, $Aborted$);
19I:      continue; // Transaction head.tx has committed/aborted $\Rightarrow$ Check head.tx.status one more time (line 2I).
20I:    end if
21I:  end if
22I: end for
23I: $newLoc.tx \leftarrow T$;
24I: SAC($newLoc.next$, $\perp$); // Store-and-clear
```

Algorithm 13 OPENW(*T*: Transaction; *O*: TMObj): Open a transactional memory object for write by a thread p_i

```
Output: reference to a data object if succeeds, or $\perp$.
1W: $newLoc \leftarrow$ new Locator;
2W: while true do
3W:   $head \leftarrow$ FINDHEAD($O$); // Find the head of O's list.
4W:   if INITLOC($newLoc, head, T$) $= \perp$ then
5W:     return $\perp$;
6W:   end if
7W:   LSA_OPEN($T, O$, "Write"); // LSA's OPEN procedure.
8W:   if $T.status = Aborted$ then
9W:     return $\perp$; // Performance (not correctness): Don't add newLoc to O if T has aborted due to, for instance, LSA_OPEN.
10W:  end if
11W:  if TFAS($head.next, newLoc$) $\neq \perp$ then
12W:    continue; // Another locator has been appended $\Rightarrow$ Find the head again
13W:  else
14W:    $oldLoc = O[i]$;
15W:    $O[i] \leftarrow (newLoc, newLoc.cts)$; // Atomic assignment; $p_i$'s old locator is unlinked from O.
16W:    SAC($oldLoc.next$, $\perp$); // oldLoc may be in the chain of a sleeping thread $\Rightarrow$ Stop the chain here
17W:    for each item $L_j$ in $O$ such that $L_j.ts < O[i].ts$ do
18W:      SAC($L_j.ptr.next$, $\perp$) // Reset the next pointer of the obsolete locator
19W:    end for
20W:    return $newLoc.new$;
21W:  end if
22W: end while
```

Lemma 3. *A locator L_i with timestamp cts_i does not have any links/references to another locator L_j with a lower timestamp $cts_j < cts_i$.*

Lemma 4. *The locator returned by* FINDHEAD(O) *(Algorithm 10) is the head H of O's locator list at the time-point* FINDHEAD *found $H.next = \perp$ (line 5F).*

Lemma 5. (Lock-freedom) FINDHEAD(O) *will certainly return the head of O's locator list after at most N repeat-until iterations unless a concurrent thread has completed a transaction and subsequently has started a new one, where N is the number of concurrent threads updating O.*

Since NBFEB-STM uses the lazy snapshot algorithm LSA [17], the second correctness criterion *Precluding inconsistent views* will follow if we can prove that the LSA algorithm is correctly integrated into NBFEB-STM.

Lemma 6. *The versions kept in N locators $O[j].ptr, 1 \leq j \leq N$, for each object O are enough for checking the validity of a transaction T using the LSA algorithm [17].*

Lemma 7. *The number of versions available for each object in NBFEB-STM is up to $(N+1)$, where N is the number of threads.*

Definition 1. *The* value *of a locator L is either $L.new$ if $L.tx.status = Committed$, or $L.old$ otherwise.*

Lemma 8. *In each O's locator list, the old value $L'.old$ of a locator L' is not older than the value of its previous locator* [10] *L.*

Lemma 9. *For each object O, there are at most $4N$ locators that cannot be reclaimed by the garbage collector at any time-point, where N is the number of update threads.*

Theorem 1. (Space complexity) *The space complexity of an object updated by N threads in NBFEB-STM is $\Theta(N)$, the optimal.*

Lemma 10. (Contention reduction) *NBFEB-STM has a lower contention level than CAS-based STMs.*

5 Conclusions and Future Work

We have introduced a new non-blocking full/empty bit primitive called NB-FEB, as a promising synchronization primitive for parallel programming on manycore architectures. We have provided a theoretical treatment of the primitive to support our claim that it is a promising primitive to consider for further research. Particularly, we have proven that i) it is universal, ii) it is combinable and thus, based on previous experimental results, it would be scalable in practice. In order to prove that it can be deployed easily as a general-purpose synchronization primitive, we have shown how to construct a non-blocking software transactional memory system NBFEB-STM with optimal space complexity using this primitive. NBFEB-STM can be used as a building block to implement concurrent algorithms conveniently.

Although the combinability makes NB-FEB promising for manycore architectures where high-contention executions are expected more frequent, experimental work is needed for future research to clearly identify the applications and system settings where NB-FEB is faster/slower than CAS or LL/SC.

Acknowledgments The authors wish to thank the anonymous reviewers for their helpful and thorough comments on the earlier version of this paper. Phuong Ha's and Otto Anshus's work was supported by the Norwegian Research Council (grant numbers 159936/V30 and 155550/420). Philippas Tsigas's work was supported by the Swedish Research Council (VR) (grant number 37252706).

[10] A locator L is a *previous* locator of a locator L' if starting from L we can reach L' by following *next* pointers.

References

1. 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.
2. K. Asanovic and et al. The landscape of parallel computing research: A view from Berkeley. *TR No. UCB/EECS-2006-183, Univ. of California, Berkeley*, 2006.
3. G. E. Blelloch, P. B. Gibbons, and S. H. Vardhan. Combinable memory-block transactions. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 23–34, 2008.
4. A. Gottlieb and et al. The NYU Ultracomputer—designing a MIMD, shared-memory parallel machine (extended abstract). In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 27–42, 1982.
5. R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 303–323, 2005.
6. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.
7. P. H. Ha, P. Tsigas, and O. J. Anshus. Nb-feb: An easy-to-use and scalable universal synchronization primitive for parallel programming. *TR No. CS-2008-69, Univ. of Tromsø, Norway*, 2008. http://www.cs.uit.no/~phuong/nbfeb_tr.pdf.
8. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the ACM Conf. on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.
9. M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, Jan. 1991.
10. M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.
11. C. P. Kruskal and et al. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.
12. V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 354–368, 2005.
13. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.1.* NVIDIA Corporation, 2007.
14. G. F. Pfister and et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *ICPP*, pages 764–771, 1985.
15. S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 159–175, 1989.
16. R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *Proc. of the Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 177–188, 2004.
17. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 284–298, 2006.
18. W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 240–248, 2005.
19. S. Sridharan, A. Rodrigues, and P. Kogge. Evaluating synchronization techniques for light-weight multithreaded/multicore architectures. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 57–58, 2007.