Wait-Free Universal Constructions that ensure Timestamp-Ignoring Disjoint-Access Parallelism

Faith Ellen
University of Toronto
faith@cs.toronto.edu

Panagiota Fatourou *
University of Crete & FORTH-ICS
faturu@csd.uoc.gr

Eleftherios Kosmas †
University of Crete & FORTH-ICS
ekosmas@csd.uoc.gr

Alessia Milani
University of Bordeaux
milani@labri.fr

Corentin Travers
University of Bordeaux
travers@labri.fr

Abstract

A *universal construction* is a method to execute sequential code in an asynchronous shared-memory system. To ensure fault-tolerance and enhance performance, universal constructions are designed to be wait-free and disjoint-access-parallel.

In a previous paper we proved that no universal construction can ensure both wait-freedom and disjoint-access parallelism. To circumvent this impossibility, while still achieving enhanced parallelism, we propose a weaker version of disjoint-access parallelism, called *timestamp-ignoring disjoint-access parallelism.* It allows two operations to access a common timestamp object, even if they are working on disjoint parts of a data structure.We present a universal construction that ensures wait-freedom and *timestamp-ignoring disjoint-access parallelism.*

1 Introduction

The dominance of multicore machines has led to an increasing need for easy ways to develop parallel code. Several parallel programming paradigms have evolved to address this need. *Transactional Memory* (TM) is an important example. It enables (appropriately-enhanced) pieces of sequential code to be executed in a concurrent environment. The goal of a *universal construction* is the same. It supports a single operation, called PERFORM, which takes as parameters a piece of sequential code and a list of input arguments for this code. The algorithm that implements PERFORM applies a sequence of operations, called *primitives*, on *base objects* provided by the system to simulate the execution of the piece of sequential code in a concurrent environment. We say that each instance of PERFORM simulates the execution of an *operation*, described by the sequential code passed to it.

We are interested in universal constructions that satisfy *wait-freedom* [11], a strong progress condition which requires that each process finishes the execution of its operation in a finite number

*Supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

†Supported by the project "IRAKLITOS II - University of Crete" of the Operational Programme for Education and Lifelong Learning 2007 - 2013 (E.P.E.D.V.M.) of the NSRF (2007 - 2013), co-funded by the European Union (European Social Fund) and National Resources.

of steps, no matter what speed it has relative to other processes and despite the failure of other processes. To enhance parallelism, it is desirable that a universal construction also exhibits a property known as *disjoint-access parallelism*, which says that operations working on different parts of a data structure do not interfere with one another.

Since its original appearance [13], disjoint-access parallelism has been extensively studied [2, 3] and many variants of it have been considered [1, 4, 10]. Most of these definitions employ the concept of a conflict graph of an execution interval. The *execution interval* of an operation is a sequence of consecutive steps taken by the processes that starts with the first step of the call to PERFORM corresponding to this operation and terminates when this call returns. Two operations *overlap* if the call to PERFORM for one of them occurs during the execution interval of the other.

The nodes of a conflict graph of an execution interval I represent the operations whose execution intervals overlap with I and an edge connects two operations in the graph if these operations access at least one common data item. Two processes *contend* on a base object if they both access it and at least one of these accesses attempts to modify the object. Strong versions of disjoint-access parallelism [10] require that the execution of two operations cannot contend on the same base object if the operations are not connected by an edge in the conflict graph, whereas weak versions [4] require only that there is a path between the two operations in the graph.

In [7] we proved that a universal construction cannot be both wait-free and feeble disjoint-access parallel. *Feeble disjoint-access parallelism* is weaker than all previously proposed definitions of disjoint-access parallelism so their result holds even if feeble disjoint-access parallelism is replaced with any previous definition. To prove the impossibility result, we employed a singly-linked unsorted list of integers supporting the operations APPEND(v), which appends a node with value v to the end of the list, and SEARCH(v), which searches the list for v starting from the first element of the list. We proved that, in any implementation resulting from the application of a universal construction to this data structure, there is an execution of SEARCH that never terminates. We also showed that this impossibility result can be circumvented by restricting attention to data structures whose operations can each only access a bounded number of different data items.

In this paper, we show that a natural relaxation of the definition of disjoint-access parallelism can overcome the impossibility result without this restriction. Specifically, we define a variant of disjoint-access parallelism, called *timestamp-ignoring* disjoint-access parallelism. It is similar to classical disjoint-access parallelism [4], but allows multiple operations to access a wait-free static timestamp object, even though the sets of data items that the operations access do not intersect. A *(static) timestamp object* [8] supports one operation : `getTimestamp`(), which returns a timestamp from a universe U with a binary relation $<$ such that $t < t'$ if an instance of `getTimestamp`() that returned t finished before an instance of `getTimestamp`() that returned t' began. A wait-free timestamp object can be easily implemented with a *fetch&increment* object or a shared global clock.

Definition 1 *A universal construction is* timestamp-ignoring *disjoint-access parallel if, in every execution, any two operations op and op′ that contend on some base object, other than the timestamp object, have a path between them in the conflict graph of the minimum execution interval containing their executions intervals.*

Several examples of algorithms that ensure timestamp-ignoring disjoint-access parallelism can be found in the literature. For instance, several well-known transactional memory systems [6, 14, 16] assign timestamps to transactions. Each transaction may then use its timestamp (as well as the timestamps of other transactions) to resolve conflicts and/or determine whether the data items it has read are consistent. If the access to the global timestamp object is not taken into consideration, some of these algorithms are disjoint access parallel (e.g. [6],[14] and [9]). However, none of these algorithms are wait-free. The definition of *timestamp-ignoring* disjoint-access parallelism can be motivated by the existence of these algorithms. This definition allows operations operating on different parts of the simulated data structure to proceed in parallel without any interference, except for accesses to the timestamp object. If the `getTimestamp`() operation never attempts to modify the timestamp object, for example, when it is implemented from a shared global clock that increments automically, then timestamp-ignoring disjoint-access parallelism is the same as disjoint-access parallelism.

An *entry point* to a data structure is any data item passed as input to an instance of an operation on the data structure. In the example of the linked-list used to prove the impossibility result in [7], the entry point for SEARCH is *first* which is the pointer to the first element of the list, and the entry point for APPEND is *last* which is a pointer to the last element of the list. Data items, such as *first* and *last*, that exist from the beginning of the execution are called *static*. Different instances of the same operation can have different entry points. For example, different entry points can be created by having a process store a pointer to some node of the data structure in a persistent local variable each time it executes an operation, and pass this pointer as input to the next operation it executes.

In Section 2, we present an algorithm which shows that the impossibility result in [7] does not hold for universal constructions that ensure timestamp-ignoring disjoint-access parallelism and wait-freedom.

2 The TI-DAP-UC Universal Construction

We present TI-DAP-UC, a universal construction that ensures wait-freedom and timestamp-ignoring disjoint-access parallelism, provided that the number of entry points in the data structure is bounded.

It is an extension of the universal construction DAP-UC presented in [7]. The additions to DAP-UC are highlighted in the code (Figures 1 and 2). For clarity, we first provide a brief description of the way DAP-UC works and then explain how DAP-UC can be enhanced to get TI-DAP-UC.

For each operation *op* it executes, DAP-UC allocates a shared record where it stores information about the operation. When a process p wants to execute an operation *op*, p starts by executing its *simulation phase* where it locally simulates the execution of *op*'s instructions without modifying the shared representation of the simulated state. Specifically, p maintains a local dictionary in which it stores the information about every data item it accesses while simulating *op*. DAP-UC also maintains a data record for each data item x. The first time *op* accesses x, it makes an announcement by writing appropriate information in x's data record. It also detects conflicts with other operations that are accessing x by reading this data record. Conflicts are resolved using a simple priority scheme in which operations invoked by processes with lower ids have higher priority. Suppose an operation *op* executed by p accesses the same data item as another operation op' executed by p'. If the process that invoked op' has higher priority than the process that invoked *op*, then p helps p' complete op' before it continues with the execution of *op*. Otherwise p causes all processes executing op' to restart and the process that invoked *op* will help complete op' (once the execution of *op* is complete) before invoking a new operation. These actions guarantee that processes never starve.

After locally simulating the instructions of *op*, p (or any helper of *op*) enters the *modifying phase* of *op*. During this phase, one of the local dictionaries of the helpers of *op* becomes shared. All helpers of *op* then use this dictionary and apply the modifications listed in it, so that all apply the same updates for *op*. This ensures consistency.

DAP-UC does not ensure wait-freedom when it is applied to data structures on which each operation can access an unbounded number of different data items. For example, in a singly-linked list, suppose a process q repeatedly appends new data items at the end of the list. The steps of another process p doing a SEARCH for an element that is not in the list can be interleaved with the steps of this execution so that q cannot distinguish it from its own infinite solo execution. Thus q never helps p terminate its SEARCH. Moreover, the SEARCH by process p cannot terminate because it cannot determine which nodes were in the list when it was invoked.

To overcome this limitation, TI-DAP-UC enhances DAP-UC in the following ways. When p invokes an operation *op*, it acquires a new timestamp by calling `getTimestamp`. The timestamp and all entry points of *op* are stored in the data record v_x of each data item x created by *op*. Static data items have timestamp 0 and entry point *null*. The first time *op* accesses a data item x, it announces itself in v_x and then checks whether the timestamp of x is larger than the timestamp of *op*. If so, the execution interval of *op* overlaps with the execution interval of the operation op' that created x, and *op* announces itself in the data record of each entry point to the data structure used by op'. Any successive operation that uses any one of these entry points will detect a conflict with *op* and help it to complete, in accordance with the priority scheme used in DAP-UC. We assume an upper bound on the number of entry points to the data structure. Therefore, an operation *op* accesses a finite number of dynamic data items before it is announced in the data record of each entry point used by each operation that creates a data item that *op* accesses. Each operation invoked after this point will either not contend with *op* or will help *op*, if it is not yet completed.

In the singly-linked list, suppose a SEARCH accesses a data item that was created by an APPEND operation op', which was invoked after the SEARCH. Then the SEARCH is announced in the data record, v_{last}, for the pointer to the last element in the list. Hence, the next APPEND invoked by each process q will help the SEARCH to complete, if the SEARCH is still in progress.

Finally, we prove that our algorithm does not violate timestamp-ignoring disjoint-access parallelism. The difficult case is when *op* is an operation that accesses a data item x created by an operation op' with a larger timestamp. Then *op* announces itself in the data records of the entry points used by op'. Let op'' be any operation that accesses one of these entry points y. Because op' is concurrent with *op*, its execution interval overlaps the minimum execution interval containing the execution intervals of *op* and op''. Thus, op' belongs to CG. Since op' accesses both y and x, there is an edge between op' and *op* and an edge between op' and op'' in CG. Thus, there is a path between *op* and op'' in CG.

Theorem 2 TI-DAP-UC *produces timestamp-ignoring disjoint-access parallel, wait-free implementations when applied to data structures with a bounded number of entry points.*

3 Discussion

In this paper, we have proposed a new version of disjoint-access parallelism, timestamp-ignoring disjoint-access parallelism, and a universal construction that ensures wait-freedom and this relaxed version of disjoint-access parallelism for unbounded data structures.

The universal construction proposed in this paper requires $\Theta(n)$ space overhead per data item. It is an open problem whether a more efficient universal construction can be designed.

Finally, it may be of interest to study other relaxations of disjoint-access parallelism. For example, S-ignoring disjoint-access parallelism, where S is a set of base objects (of possibly different types), ensures that disjoint-access parallelism is guaranteed when accessing all objects other than those in the set S.

References

[1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling Multi-object Operations (Extended Abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1997, pages 111–120.

[2] H. Attiya and E. Dagan. Universal operations: Unary versus Binary. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1996, pages 223–232.

[3] H. Attiya and E. Hillel. Built-in Coloring for Highly-Concurrent Doubly-Linked Lists. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006, pages 31–45.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent Limitations on Disjoint-Access Parallel Implementations of Transactional Memory. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009, pages 69–78.

[5] V. Bushkov, R. Guerraoui, and M. Kapalka. On the Liveness of Transactional Memory. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, 2012, pages 9–18.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006, pages 194–208.

[7] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2012, pages 115–124.

[8] F. Ellen, P. Fatourou, and E. Ruppert. The Space Complexity of Unbounded Timestamps. In *Proceedings of the 21st International Conference on Distributed Computing (DISC)*, 2007, pages 223–237.

[9] P. Felber, C. Christof Fetzer, P. Marlier, and T. Riegel, Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12), 2010, pages 1793–1807.

[10] R. Guerraoui and M. Kapalka. On Obstruction-Free Transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008, pages 304–313.

[11] M. Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languanges and Systems*, 13(1), 1991, pages 124–149.

[12] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990, pages 463–492.

[13] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)* 1994, pages 151–160.

[14] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006 , pages 284–298.

[15] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM symposium on Principles of Distributed Computing (PODC)*, 1995, pages 204–213.

[16] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. newblock In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006 , pages 179–193.

```
type varrec
    value val
    tmval tm
    set of ptr to varrec pvar
    ptr to oprec A[1..n]

type statrec
    {⟨simulating⟩,
     ⟨restart, ptr to oprec restartedby⟩,
     ⟨modifying, ptr to dictionary of dictrec changes, value output⟩
     ⟨done⟩
    } status

type oprec
    code  program
    process id  owner
    value input
    value output
    tmval tm
    set of ptr to varrec pentry
    ptr to statrec status
    ptr to oprec tohelp[1..n]

type dictrec
    ptr to varrec key
    value newval

Announce(opptr, x) by process p:
    q := opptr → owner
    LL(x → A[q])
    if ¬ VL(opptr → status) then return
    SC(x → A[q], opptr)
    LL(x → A[q])
    if ¬VL(opptr → status) then return
    SC(x → A[q], opptr)
    return

Conflicts(opptr, x) by process p:
    for p′ := 1 to n excluding opptr → owner do
        opptr′ := LL(x → A[p′])
        if (opptr′ ≠ nil) then                              /* possible conflict between op and op′ */
            opstatus′ := LL(oppptr′ → status)
            if ¬VL(opptr → status) then return
            if (opstatus′ = ⟨modifying, changes, output⟩)
                then Help(opptr′)
            else if (opstatus′ = ⟨simulating⟩) then
                if (opptr → owner < p′) then               /* op has higher priority than op′, restart op′ */
                    opptr → tohelp[p′] := opptr′
                    if ¬VL(opptr → status) then return
                    SC(opptr′ → status, ⟨restart, opptr⟩)
                    if (LL(oppptr′ → status) = ⟨modifying, changes, output⟩) then
                        Help(opptr′)
                else Help(opptr′)                           /* opptr → owner > p′ */
    return
```

Figure 1: Type definitions and the code of Announce and Conflicts of TI-DAP-UC.

```
value Perform(prog, input) by process p:
    opptr := pointer to a new oprec record
    opptr → program := prog, opptr → input := input, opptr → output := ⊥, opptr → owner := p
    | opptr → tm := getTimestamp(), opptr → pentry := input.entry |
    opptr → status := simulating, opptr → tophelp[1..n] := [nil, . . . , nil]
    Help(opptr)                                          /* p helps its own operation */
    for p′ := 1 to n excluding p do   /* p helps operations that have been restarted by its operation op */
        if (opptr → tohelp[p′] ≠ nil) then Help(opptr → tohelp[p′])
    return(opptr → output)

Help(opptr) by process p:
    opstatus := LL(opptr → status)
    while (opstatus ≠ done)
        if opstatus = ⟨restart, opptr′⟩ then                          /* op′ has restarted op */
            Help(opptr′)                                               /* first help op′ */
            SC(opptr → status, ⟨simulating⟩)       /* try to change the status of op back to simulating */
            opstatus := LL(opptr → status)
        if opstatus = ⟨simulating⟩ then                          /* start a new simulation phase */
            dict := pointer to a new empty dictionary of dictrec records
                                                              /* to store the values of the data items */
            ins := the first instruction in opptr → program
            while ins ≠ return(v)                                 /* simulate instruction ins of op */
                if ins is (WriteDI(x, v) or ReadDI(x)) and (there is no dictrec with key x in dict)
                then                                          /* first access of x by this attempt of op */
                    | if (opptr → tm < x → tm) then                        |
                    |         for each y in x → pvar do Announce(opptr, y) |
                    Announce(opptr, x)                               /* announce that op is accessing x */
                    Conflicts(opptr, x)              /* possibly, help or restart other operations accessing x */
                    if ins = ReadDI(x) then val_x := x → val  else val_x := v      /* ins is a write to x of v */
                    add new dictrec ⟨x, val_x⟩ to dict                          /* create a local copy of x */
                else if ins is CreateDI() then
                    x := pointer to a new varrec record
                    | x → tm := opptr → tm,  x → pvar := opptr → pentry |
                    x → A[1..n] := [nil, . . . , nil]
                    add new dictrec ⟨x, nil⟩ to dict
                else              /* ins is WriteDI(x, v) or ReadDI(x) and there is a dictrec with key x in dict */
                                         /* or ins is not a WriteDI(), ReadDI() or CreateDI() instruction */
                    execute ins, using/changing the value in the appropriate entry of dict if necessary
                if ¬VL(opptr → status) then break                       /* end of the simulation of ins */
                ins := next instruction of opptr → program
            /* end while */
            if ins is return(v) then                                           /* v may be empty */
                SC(opptr → status,⟨modifying, dict, v⟩)          /* try to change status of op to modifying */
                      /* successful iff simulation is over and status of op not changed since beginning of simulation */
            opstatus := LL(opptr → status)
        if opstatus = ⟨modifying, changes, out⟩ then
            opptr → outputs := out
            for each dictrec ⟨x, v⟩ in the dictionary pointed to by changes do
                LL(x → val)                                              /* try to make writes visible */
                if ¬VL(opptr → status) then return                           /* opptr → status = done */
                SC(x → val, v)
                LL(x → val)
                if ¬VL(opptr → status) then return                           /* opptr → status = done */
                SC(x → val, v)
            /* end for */
            SC(opptr → status, done)
            opstatus := LL(opptr → status)
    /* end while */
    return
```

Figure 2: The code of Perform and Help of TI-DAP-UC.