The Weakest Failure Detectors to Boost Obstruction-Freedom

Rachid Guerraoui[1,2], Michał Kapałka[2], and Petr Kouznetsov[3]

[1] Computer Science and Artificial Intelligence Laboratory, MIT
[2] School of Computer and Communication Sciences, EPFL
[3] Max Planck Institute for Software Systems

Abstract. This paper determines necessary and sufficient conditions to implement *wait-free* and *non-blocking* contention managers in a shared memory system. The necessary conditions hold even when universal objects (like compare-and-swap) or random oracles are available, whereas the sufficient ones assume only registers.
We show that failure detector $\Diamond\mathcal{P}$ is the weakest to convert any obstruction-free algorithm into a wait-free one, and Ω^*, a new failure detector which we introduce in this paper, and which is strictly weaker than $\Diamond\mathcal{P}$ but strictly stronger than Ω, is the weakest to convert any obstruction-free algorithm into a non-blocking one.

1 Introduction

Multiprocessor systems are becoming more and more common nowadays. Multithreading thus becomes the norm and studying scalable and efficient synchronization methods is essential, for traditional locking-based techniques do not scale and may induce priority inversion, deadlock and fault-tolerance issues when a large number of threads is involved.

Wait-free synchronization algorithms [1] circumvent the issues of locking and guarantee individual progress even in presence of high contention. Wait-freedom is a liveness property which stipulates that every process completes every operation in a finite number of its own steps, regardless of the status of other processes, i.e., contending or even crashed. Ideal synchronization algorithms would ensure *linearizability* [2,3], a safety property which provides the illusion of instantaneous operation executions, together with wait-freedom.

Alternatively, a liveness property called *non-blockingness*[4] may be considered instead of wait-freedom. Non-blockingness guarantees global progress, i.e., that some process will complete an operation in a finite number of steps, regardless of the behavior of other processes. Non-blockingness is weaker than wait-freedom as it does not prevent some processes from starvation.

[4] The term *non-blocking* is defined here in the traditional way [1]: "some process will complete its operation in a finite number of steps, regardless of the relative execution speeds of the processes." This term is sometimes confused with the term *lock-free*. Note that non-blocking implementations provide a weaker liveness guarantee than wait-free implementations.

Wait-free and non-blocking algorithms are, however, notoriously difficult to design [4,5], especially with the practical goal to be fast in low contention scenarios, which are usually considered the most common in practice. An appealing principle to reduce this difficulty consists in separating two concerns of a synchronization algorithm: (1) ensuring linearizability with a minimal conditional progress guarantee, and (2) boosting progress. More specifically, the idea is to focus on algorithms that ensure linearizability together with a weak liveness property called *obstruction-freedom* [6], and then combine these algorithms with separate generic oracles that boost progress, called *contention managers* [7,8,9,10]. This separation lies at the heart of modern (obstruction-free) software transactional memory (STM) frameworks [7].

With obstruction-free (or OF, for short) algorithms, progress is ensured only for every process that executes in isolation for sufficiently long time. In presence of high contention, however, OF algorithms can livelock, preventing any process from terminating. Contention managers are used precisely to cope with such scenarios. When queried by a process executing an OF algorithm, a contention manager can delay the process for some time in order to boost the progress of other processes. The contention manager can neither share objects with the OF algorithm, nor return results on its behalf. If it did, the contention manager could peril the safety of the OF algorithm, hampering the overall separation of concerns principle.

In short, the goal of a contention manager is to provide processes with enough time without contention so that they can complete their operations. In its simplest form, a contention manager can be a randomized back-off protocol. More sophisticated contention management strategies have been experimented in practice [8,9,11]. Precisely because they are entirely devoted to progress, they can be combined or changed on the fly [10]. Most previous strategies were *pragmatic*, with no aim to provide *worst case guarantees*. In this paper we focus on contention managers that provide such guarantees. More specifically, we study contention managers that convert any OF algorithm into a non-blocking or wait-free one, and which we call, respectively, *non-blocking* or *wait-free* contention managers.

Two wait-free contention managers have recently been proposed [12,13]. Both rely on timing assumptions to detect processes that fail in the middle of their operations. This suggests that *some* information about failures might inherently be needed by any wait-free contention manager. But this is not entirely clear because, in principle, a contention manager could also use randomization to schedule processes, or even powerful synchronization primitives like compare-and-swap, which is known to be *universal*, i.e., able to wait-free implement any other object [1]. In the parlance of [14], we would like to determine whether a *failure detector* is actually needed to implement a contention manager with worst case guarantees, and if it is, what is the *weakest* one [15]. Besides the theoretical interest, determining the minimal conditions under which a contention manager can ensure certain guarantees is, we believe, of practical relevance, for this might help portability and optimization.

We show that the eventually perfect failure detector $\Diamond\mathcal{P}$ [14] is the weakest to implement a wait-free contention manager.[5] We also introduce a failure detector Ω^*, which we show is the weakest to implement a non-blocking contention manager. Failure detector Ω^* is strictly weaker than $\Diamond\mathcal{P}$, and strictly stronger than failure detector Ω [15], known to be the weakest to wait-free implement the (universal) consensus object [1].[6]

It might be surprising that Ω is not sufficient to implement a wait-free or even a non-blocking contention manager. For example, the seminal Paxos algorithm [16] uses Ω to transform an OF implementation of consensus into a wait-free one. Each process that is eventually elected a leader by Ω is given enough time to run alone, reach a decision and communicate it to the others. This approach does not help, however, if we want to make sure that processes make progress regardless of the actual (possibly long-lived) object and its OF implementation. Intuitively, the leader elected by Ω may have no operation to perform while other processes may livelock forever. Because a contention manager cannot make processes help each other, the output of Ω is not sufficient: this is so even if randomized oracles or universal objects are available. Intuitively, wait-free contention managers need a failure detector that would take care of *every* non-crashed process with a pending operation so that the process can run alone for sufficiently long time. As for non-blocking contention managers, at least *one* process that never crashes, among the ones with pending operations, should be given enough time to run alone.

The paper is organized as follows. Section 2 presents our system model and formally defines wait-free and non-blocking contention managers. These definitions are, we believe, contributions in their own rights, for they capture precisely the interaction between a contention manager and an obstruction-free algorithm. In Sect. 3 and 4, we prove our weakest failure detector results. In each case, we first present (necessary part) a *reduction* algorithm [15] that *extracts* the output of failure detector Ω^* (respectively $\Diamond\mathcal{P}$) using a non-blocking (respectively wait-free) contention manager implementation. When devising our reduction algorithms, we do not restrict what objects (or random oracles) can be used by the contention manager or the OF algorithm. Then (sufficient part), we present algorithms that implement the contention managers using the failure detectors and registers. These algorithms are devised with the sole purpose of proving our sufficiency claims. We do not seek to minimize the overhead of the interaction between the OF algorithm and the contention manager, nor do we discuss how the failure detector can itself be implemented with little synchrony assumptions and minimal overhead, unlike the transformations presented in [12]. However, as we show in [17], our algorithms can be easily extended to meet these challenges. The proofs of a few minor results are omitted due to the space limitations and can be found in the full version of the paper [18].

[5] $\Diamond\mathcal{P}$ ensures that eventually: (1) every failure is detected by every correct (i.e., non-faulty) process and (2) there is no false detection.

[6] Ω ensures that eventually all correct (i.e., non-faulty) processes elect the same correct process as their leader.

2 Preliminaries

Processes and Failure Detectors. We consider a set of n processes $\Pi = \{p_1, \ldots, p_n\}$ in a shared memory system [1,19]. A process executes the (possibly randomized) algorithm assigned to it, until the process *crashes* (*fails*) and stops executing any action. We assume the existence of a global discrete clock that is, however, inaccessible to the processes. We say that a process is *correct* if it never crashes. We say that process p_i is *alive* at time t if p_i has not crashed by time t.

A *failure detector* [14,15] is a distributed oracle that provides every process with some information about failures. The output of a failure detector depends only on which and when processes fail, and not on computations being performed by the processes. A process p_i queries a failure detector $\mathcal{D}$ by accessing local variable $\mathcal{D}$-$output_i$—the output of the module of $\mathcal{D}$ at process p_i. Failure detectors can be partially ordered according to the amount of information about failures they provide. A failure detector $\mathcal{D}$ is *weaker than a failure detector* $\mathcal{D}'$, and we write $\mathcal{D} \preceq \mathcal{D}'$, if there exists an algorithm (called a *reduction* algorithm) that transforms $\mathcal{D}'$ into $\mathcal{D}$. If $\mathcal{D} \preceq \mathcal{D}'$ but $\mathcal{D}' \not\preceq \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$, and we write $\mathcal{D} \prec \mathcal{D}'$.

Base and High-Level Objects. Processes communicate by invoking primitive operations (which we will call *instructions*) on *base* shared objects and seek to implement the *operations* of a *high-level* shared object O. Object O is in turn used by an application, as a high-level inter-process communication mechanism. We call invocation and response events of a high-level operation *op* on the implemented object O *application events* and denote them by, respectively, $inv(op)$ and $ret(op)$ (or $inv_i(op)$ and $ret_i(op)$ at a process p_i).

An *implementation* of O is a distributed algorithm that specifies, for every process p_i and every operation *op* of O, the sequences of *steps* that p_i should take in order to complete *op*. Process p_i *completes* operation *op* when p_i returns from *op*. Every process p_i may complete any number of operations but, at any point in time, at most one operation *op* can be *pending* (started and not yet completed) at p_i.

We consider implementations of O that combine a sub-protocol that ensures a minimal liveness property, called *obstruction-freedom*, with a sub-protocol that boosts this liveness guarantee. The former is called an *obstruction-free (OF)* algorithm A and the latter a *contention manager* CM. We focus on *linearizable* [2,3] implementations of O: every operation appears to the application as if it took effect instantaneously between its invocation and its return. An implementation of O involves two categories of steps executed by any process p_i: those (executed on behalf) of CM and those (executed on behalf) of A. In each step, a process p_i either executes an instruction on a base shared object or (in case p_i executes a step on behalf of CM) queries a failure detector.

Obstruction-freedom [6,7] stipulates that if a process that invokes an operation *op* on object O and from some point in time executes steps of A alone[7], then it eventually completes *op*. *Non-blockingness* stipulates that if some correct process never completes an invoked operation, then some other process completes infinitely many operations. *Wait-freedom* [1] ensures that every correct process that invokes an operation eventually returns from the operation.

Interaction Between Modules. OF algorithm A, executed by any process p_i, communicates with contention manager CM via *calls* try_i and $resign_i$ implemented by CM (see Fig. 1). Process p_i invokes try_i just after p_i starts an operation, and also later (even several times before p_i completes the operation) to signal possible contention. Process p_i invokes $resign_i$ just before returning from an operation, and always eventually returns from this call (or crashes). Both calls, try_i and $resign_i$, return *ok*.[8]

We denote by $B(A)$ and $B(CM)$ the sets of base shared objects, always *disjoint*, that can be possibly accessed by steps of, respectively, A and CM, in every execution, by every process. Calls *try* and *resign* are thus the only means by which A and CM interact. The events corresponding to invocations of, and responses from, *try* and *resign* are called *cm-events*. We denote by try_i^{inv} and $resign_i^{\text{inv}}$ an invocation of call try_i and $resign_i$, respectively (at process p_i), and by try_i^{ret} and $resign_i^{\text{ret}}$—the corresponding responses.

Executions and Histories. An *execution* of an OF algorithm A combined with a contention manager CM is a sequence of *events* that include steps of A, steps of CM, cm-events and application events. Every event in an execution is associated with a unique time at which the event took place. Every execution e induces a *history* $H(e)$ that includes only application events (invocations and responses of high-level operations). The corresponding *CM-history* $H_{\text{CM}}(e)$ is the subsequence of e containing only application events and cm-events of the execution, and the corresponding *OF-history* $H_{\text{OF}}(e)$ is the subsequence of e containing only application events, cm-events, and steps of A. For a sequence s of events, $s|i$ denotes the subsequence of s containing only events at process p_i.

We say that a process p_i is *blocked* at time t in an execution e if (1) p_i is alive at time t, and (2) the latest event in $H_{\text{CM}}(e)|i$ that occurred before t is try_i^{inv} or $resign_i^{\text{inv}}$. A process p_i is *busy* at time t in e if (1) p_i is alive at time t, and (2) the latest event in $H_{\text{CM}}(e)|i$ that occurred before t is try_i^{ret}. We say that a process p_i is *active* at t in e if p_i is either busy or blocked at time t in e. We say that a process p_i is *idle* at time t in e if p_i is not active at t in e.[9] A process *resigns* when it invokes *resign* on a contention manager.

[7] I.e., without encountering *step contention* [20].

[8] An example OF algorithm that uses this model of interaction with a contention manager is presented in [18]. A discussion about overhead of wait-free/non-blocking contention managers that explains when calls to *try*/*resign* can be omitted for efficiency reasons can be found in [17].

[9] Note that every process that has crashed is permanently idle.

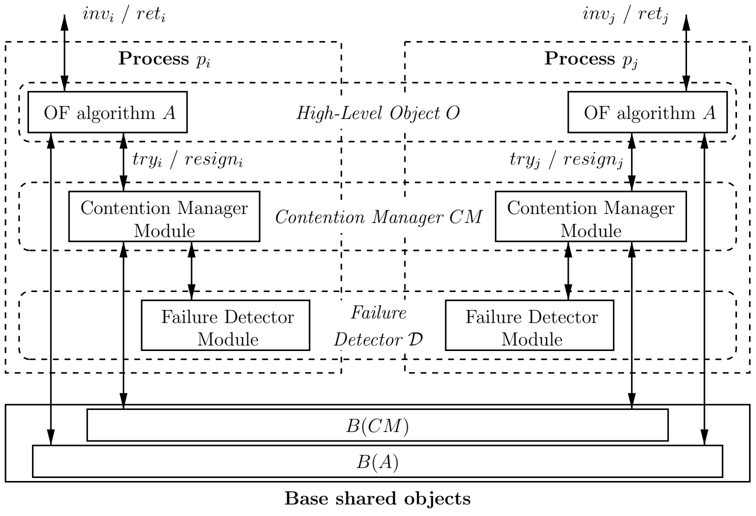

Fig. 1. The OF algorithm/contention manager interface

We say that p_i is *obstruction-free* in an interval $[t, t']$ in an execution e, if p_i is the only process that takes steps of A in $[t, t']$ in e and p_i is not blocked infinitely long in $[t, t']$ (if $t' = \infty$). We say that process p_i is *eventually obstruction-free* at time t in e if p_i is active at t or later and p_i either resigns after t or is obstruction-free in the interval $[t', \infty)$ for some $t' > t$. Note that, since algorithm A is obstruction-free, if an active process p_i is eventually obstruction-free, then p_i eventually resigns and completes its operation.

Well-Formed Executions. We impose certain restrictions on the way an OF algorithm A and a contention manager CM interact. In particular, we assume that no process takes steps of A while being blocked by CM or idle, and no process takes infinitely many steps of A without calling CM infinitely many times. Further, a process must inform CM that an operation is completed by calling *resign* before returning the response to the application.

Formally, we assume that every execution e is *well-formed*, i.e., $H(e)$ is linearizable [2,3], and, for every process p_i, (1) $H_{\mathrm{CM}}(e)|i$ is a prefix of a sequence $[op_1][op_2], \ldots$, where each $[op_k]$ has the form $inv_i(op_k)$,try_i^{inv}, try_i^{ret}, $\ldots$, try_i^{inv}, try_i^{ret}, $resign_i^{\mathrm{inv}}$, $resign_i^{\mathrm{ret}}$,$ret_i(op_k)$; (2) in $H_{\mathrm{OF}}(e)|i$, no step of A is executed when p_i is blocked or idle, (3) in $H_{\mathrm{OF}}(e)|i$, inv_i can only be followed by try_i^{inv}, and ret_i can only be preceded by $resign_i^{\mathrm{ret}}$; (4) if p_i is busy at time t in e, then at some $t' > t$, process p_i is idle or blocked. The last condition implies that every busy process p_i eventually invokes try_i (and becomes blocked), resigns or crashes. Clearly, in a well-formed execution, every process goes through the following cyclical order of modes: *idle*, *active*, *idle*, …, where each *active* period consists itself of a sequence *blocked*, *busy*, *blocked*, ….

Non-blocking Contention Manager. We say that a contention manager *CM guarantees non-blockingness for an OF algorithm A* if in each execution e of A combined with *CM* the following property is satisfied: if some correct process is active at a time t, then at some time $t' > t$ some process resigns.

A *non-blocking contention manager* guarantees non-blockingness for every OF algorithm. Intuitively, this will happen if the contention manager allows at least one active process to be obstruction-free (and busy) for sufficiently long time, so that the process can complete its operation. More precisely, we say that a contention manager *CM* is *non-blocking* if, for every OF algorithm A, in every execution of A combined with *CM* the following property is ensured at every time t:

Global Progress. If some correct process is active at t, then some correct process is eventually obstruction-free at t.

We show in [18] that a contention manager *CM* guarantees non-blockingness for every OF algorithm if and only if *CM* is non-blocking.

Wait-Free Contention Manager. We say that a contention manager *CM guarantees wait-freedom for an OF algorithm A* if in every execution e of A combined with *CM*, the following property is satisfied: if a process p_i is active at a time t, then at some time $t' > t$, p_i becomes idle. In other words, every operation executed by a correct process eventually returns.

A *wait-free contention* manager guarantees wait-freedom for every OF algorithm. Intuitively, this will happen if the contention manager makes sure that every correct active process is given “enough” time to complete its operation, regardless of how other processes behave. More precisely, a contention manager *CM* is wait-free if, for every OF algorithm A, in every execution of A combined with *CM*, the following property is ensured at every time t:[10]

Fairness. If a correct process p_i is active at t, then p_i is eventually obstruction-free at t.

We show in [18] that a contention manager *CM* guarantees wait-freedom for every OF algorithm if and only if *CM* is wait-free.

In the following, we seek to determine the *weakest* [15] failure detector $\mathcal{D}$ to implement a non-blocking (resp. wait-free) contention manager *CM*. This means that (1) $\mathcal{D}$ implements such a contention manager, i.e., there is an algorithm that implements *CM* using $\mathcal{D}$, and (2) $\mathcal{D}$ is *necessary* to implement such a contention manager, i.e., if a failure detector $\mathcal{D}'$ implements *CM*, then $\mathcal{D} \preceq \mathcal{D}'$. In our context, a reduction algorithm that transforms $\mathcal{D}'$ into $\mathcal{D}$ uses the $\mathcal{D}'$-based implementation of the corresponding contention manager as a “black box” and read-write registers.

[10] This property is ensured by wait-free contention managers from the literature [12,13].

3 Non-blocking Contention Managers

Let $S \subseteq \Pi$ be a non-empty set of processes. Failure detector Ω_S outputs, at every process, an identifier of a process (called a *leader*), such that all correct processes *in* S eventually agree on the identifier of the same *correct* process *in* S.[11]

Failure detector Ω^* is the composition $\{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$: at every process p_i, Ω^**-output*$_i$ is a tuple consisting of the outputs of failure detectors Ω_S. We position Ω^* in the hierarchy of failure detectors of [14] by showing in [18] that $\Omega \prec \Omega^* \prec \Diamond\mathcal{P}$.

To show that Ω^* is necessary to implement a non-blocking contention manager, it suffices to prove that, for every non-empty $S \subseteq \Pi$, Ω_S is necessary to implement a non-blocking contention manager. Let *CM* be a non-blocking contention manager using failure detector $\mathcal{D}$. We show that $\Omega^* \preceq \mathcal{D}$ by presenting an algorithm $T_{\mathcal{D} \rightarrow \Omega_S}$ (Algorithm 1) that, using *CM* and $\mathcal{D}$, emulates the output of Ω_S.

Algorithm 1: Extracting Ω_S from a non-blocking contention manager (code for processes from set S; others are permanently idle)

```
uses: $L$—register
initially: $\Omega_S$-$output_i \leftarrow p_i$, $L \leftarrow$ some process in $S$
  Launch two parallel tasks: $T_i$ and $F_i$

1.1 parallel task $F_i$
1.2     $\Omega_S$-$output_i \leftarrow L$

1.3 parallel task $T_i$
1.4     while true do
1.5         issue $try_i$ and wait until busy (i.e., until call $try_i$ returns)
1.6         $L \leftarrow p_i$  // announce yourself a leader
```

The algorithm works as follows. Every process $p_i \in S$ runs two parallel tasks T_i and F_i. In task T_i, process p_i periodically (1) gets blocked by *CM* after invoking try_i (line 1.5), and (2) once p_i gets busy again, announces itself a leader for set S by writing its id in L (line 1.6). In task F_i, process p_i periodically determines its leader by reading register L (line 1.2).[12]

Thus, no process ever resigns and every correct process in S is permanently active from some point in time. Intuitively, this signals a possible livelock to *CM* which has to eventually block all active processes except for one that should run obstruction-free for sufficiently long time. By Global Progress, *CM* cannot

[11] Ω_S can be seen as a restriction of the eventual leader election failure detector Ω [15] to processes in S. The definition of Ω_S resembles the notion of Γ-accurate failure detectors introduced in [21]. Clearly, Ω_Π is Ω.

[12] If a process is blocked in one task, it continues executing steps in parallel tasks.

block *all* active processes forever and so if the elected process crashes (and so becomes idle), CM lets another active process run obstruction-free. Eventually, all correct processes in S agree on the same process in S. Processes outside S are permanently idle and permanently output their own ids: they do not access CM.

This approach contains a subtlety. To make sure that there is a time after which the same correct leader in S is permanently elected by the correct processes in S, we do not allow the elected leader to resign (the output of Ω_S has to be eventually stable). This violates the assumption that processes using CM run an obstruction-free algorithm, and, thus, a priori, CM is not obliged to preserve Global Progress. However, as we show below, since CM does not "know" how much time a process executing an OF algorithm requires to complete its operation, CM has to provide some correct process with *unbounded* time to run in isolation.

Theorem 1. *Every non-blocking contention manager can be used to implement failure detector Ω^*.*

Proof. Let $S \subseteq \Pi$, $S \neq \emptyset$ and consider any execution of Algorithm 1. If S contains no correct process, then Ω_S-$output_i$ (for every process $p_i \in S$) trivially satisfies the property of Ω_S. Now assume that there is a correct process in S. We claim that CM eventually lets exactly one correct process in S run obstruction-free while blocking forever all the other processes in S.

Suppose not. We obtain an execution in which every correct process in S is allowed to be obstruction-free only for bounded periods of time. But the CM-history of this execution corresponds to an execution of some OF algorithm A combined with CM in which no active process ever completes its operation because no active process ever obtains enough time to run in isolation. Thus, no active process is eventually obstruction-free in that execution. This contradicts the assumption that CM is non-blocking.

Therefore, there is a time after which exactly one correct process $p_j \in S$ is periodically busy (others are blocked or idle forever) and, respectively, register L permanently stores the identifier of p_j. Thus, eventually, every correct process in S outputs p_j: the output of Ω_S is extracted. $\square$

We describe an implementation of a non-blocking contention manager using Ω^* and registers in Algorithm 2 (we prove its correctness in [18]). The algorithm works as follows. All active processes, upon calling *try*, participate in the leader election mechanism using Ω^* in lines 2.3–2.5. The active process p_i that is elected a leader returns from *try* and is (eventually) allowed to run obstruction-free until p_i resigns. Once p_i resigns, the processes elect another leader. Failure detector Ω^* guarantees that if an active process is elected and crashes before resigning, another active process is eventually elected.

Theorem 2. *Algorithm 2 implements a non-blocking contention manager.*

Algorithm 2: A non-blocking contention manager using $\Omega^* = \{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$

```
    uses: $T[1,\ldots,n]$—array of single-bit registers
    initially: $T[1,\ldots,n] \leftarrow \mathit{false}$
2.1 upon $\mathit{try}_i$ do
2.2     $T[i] \leftarrow \mathit{true}$
2.3     repeat
2.4         $S \leftarrow \{\, p_j \in \Pi \mid T[j] = \mathit{true}\}$
2.5     until $\Omega_S$-$\mathit{output}_i = p_i$
2.6 upon $\mathit{resign}_i$ do
2.7     $T[i] \leftarrow \mathit{false}$
```

4 Wait-Free Contention Managers

We prove here that the weakest failure detector to implement a wait-free contention manager is $\Diamond\mathcal{P}$. Failure detector $\Diamond\mathcal{P}$ [14] outputs, at each time and every process, a set of *suspected* processes. There is a time after which (1) every crashed process is permanently suspected by every correct process and (2) no correct process is ever suspected by any correct process.

We first consider a wait-free contention manager *CM* using a failure detector $\mathcal{D}$, and we exhibit a reduction algorithm $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$ (Algorithm 3) that, using *CM* and $\mathcal{D}$, emulates the output of $\Diamond\mathcal{P}$.

Algorithm 3: Extracting $\Diamond\mathcal{P}$ from a wait-free contention manager

```
     uses: $R[1,\ldots,n]$—array of registers
     initially: $\Diamond\mathcal{P}$-$\mathit{output}_i \leftarrow \Pi - \{p_i\}$, $k \leftarrow 0$, $R[i] \leftarrow 0$
       Launch $n(n-1)$ parallel instances of $\mathit{CM}$: $C_{jk}$, $j,k \in \{1,\ldots,n\}$, $j \neq k$
       Launch $2n-1$ parallel tasks: $T_{ij}$, $T_{ji}$, $j \in \{1,\ldots,n\}$, $i \neq j$, and $F_i$
3.1  parallel task $F_i$
3.2      while $\mathit{true}$ do  $R[i] \leftarrow R[i]+1$   // ‘‘heartbeat’’ signal
3.3  parallel task $T_{ij}$, $j = 1,\ldots,i-1,i+1,\ldots,n$
3.4      while $\mathit{true}$ do
3.5          $x_j \leftarrow R[j]$
3.6          $\Diamond\mathcal{P}$-$\mathit{output}_i \leftarrow \Diamond\mathcal{P}$-$\mathit{output}_i - \{p_j\}$  // stop suspecting $p_j$
3.7          issue $\mathit{try}_i^{ij}$ (in $C_{ij}$) and wait until busy
3.8          issue $\mathit{resign}_i^{ij}$ (in $C_{ij}$) and wait until idle
3.9          $\Diamond\mathcal{P}$-$\mathit{output}_i \leftarrow \Diamond\mathcal{P}$-$\mathit{output}_i \cup \{p_j\}$  // start suspecting $p_j$
3.10         wait until $R[j] > x_j$  // wait until $p_j$ takes a new step
3.11 parallel task $T_{ji}$, $j = 1,\ldots,i-1,i+1,\ldots,n$
3.12     while $\mathit{true}$ do  issue $\mathit{try}_i^{ji}$ (in $C_{ji}$) and wait until busy
```

We run several instances of CM. These instances use disjoint sets of base shared objects and do not directly interact. Basically, in each instance, only two processes are active and all other processes are idle. One of the two processes, say p_j, gets active and never resigns thereafter, while the other, say p_i, permanently alternates between being active and idle. To CM it looks like p_j is always obstructed by p_i. Thus, to guarantee wait-freedom, the instance of CM has to eventually block p_i and let p_j run obstruction-free until p_j resigns *or crashes*. Therefore, when p_i is blocked, p_i can assume that p_j is alive and when p_i is busy, p_i can suspect p_j of having crashed, until p_i eventually observes p_j's "heartbeat" signal, which p_j periodically broadcasts using a register. This ensures the properties of $\Diamond\mathcal{P}$ at process p_i, provided that p_j never resigns.

As in Sect. 3, we face the following issue. If p_j is correct, p_i will be eventually blocked forever and p_j will thus be eventually obstruction-free. Hence, in the corresponding execution, obstruction-freedom is violated, i.e., the execution cannot be produced by any OF algorithm combined with CM. One might argue then that CM is not obliged to preserve Fairness with respect to p_j. However, we show that, since CM does not "know" how much time a process executing an OF algorithm requires to complete its operation, CM has to provide p_j with *unbounded* time to run in isolation.

More precisely, the processes in Algorithm 3 run $n(n-1)$ parallel instances of CM, denoted each CM_{jk}, where $j, k \in \{1, \ldots, n\}$, $j \neq k$. We denote the events that process p_i issues in instance CM_{jk} by try_i^{jk} and $resign_i^{jk}$. Besides, every process p_i runs $2n-1$ parallel tasks: T_{ij}, T_{ji}, where $j \in \{1, \ldots, n\}$, $i \neq j$, and F_i. Every task T_{ij} executed by p_i is responsible for detecting failures of process p_j. Every task T_{ji} executed by p_i is responsible for preventing p_j from falsely suspecting p_i. In task F_i, p_i periodically writes ever-increasing "heartbeat" values in a shared register $R[i]$.

In every instance CM_{ij}, there can be only two active processes: p_i and p_j. Process p_i cyclically gets active (line 3.7) and resigns (line 3.8), and process p_j gets active once and keeps getting blocked (line 3.12). Each time before p_i gets active, p_i removes p_j from the list of suspected processes (line 3.6). Each time p_i stops being blocked, p_i starts suspecting p_j (line 3.9) and waits until p_i observes a "new" step of p_j (line 3.10). Once such a step of p_j is observed, p_i stops suspecting p_j and gets active again.

Theorem 3. *Every wait-free contention manager can be used to implement failure detector $\Diamond\mathcal{P}$.*

Proof. Consider any execution e of $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$, and let p_i be any correct process. We show that, in e, $\Diamond\mathcal{P}$-$output_i$ satisfies the properties of $\Diamond\mathcal{P}$, i.e., p_i eventually permanently suspects every non-correct process and stops suspecting every correct process. (Note that if a process p_i is not correct, then $\Diamond\mathcal{P}$-$output_i$ trivially satisfies the properties of $\Diamond\mathcal{P}$.)

Let p_j be any process distinct from p_i. Assume p_j is not correct. Thus p_i is the only correct active process in instance CM_{ij}. By the Fairness property of CM, p_i is eventually obstruction-free every time p_i becomes active, and so p_i

cannot be blocked infinitely long in line 3.7. Since there is a time after which p_j stops taking steps, eventually p_i starts suspecting p_j (line 3.9) and suspends in line 3.10, waiting until p_j takes a new step. Thus, p_i eventually suspects p_j forever.

Assume now that p_j is correct. We claim that p_i must eventually get permanently blocked so that p_j would run obstruction-free from some point in time forever. Suppose not. But then we obtain an execution in which p_i alternates between active and idle modes infinitely many times, and p_j stays active and runs obstruction-free only for bounded periods of time. But the CM-history of this execution could be produced by an execution e' of some OF algorithm combined with *CM* in which p_j never completes its operation because p_j never runs long enough in isolation. Thus, Fairness is violated in execution e' and this contradicts the assumption that *CM* is wait-free. Hence, eventually p_i gets permanently blocked in line 3.7. Since each time p_i is about to get blocked, p_i stops suspecting p_j in line 3.6, there is a time after which p_i never suspects p_j.

Thus, there is a time after which, if p_j is correct, then p_j stops being suspected by every correct process, and if p_j is non-correct, then every correct process permanently suspects p_j. $\square$

We describe an implementation of a wait-free contention manager using $\Diamond\mathcal{P}$ and registers in Algorithm 4 (we prove its correctness in [18]). The algorithm relies on a (wait-free) primitive *GetTimestamp()* that generates unique, locally increasing timestamps and makes sure that if a process gets a timestamp ts, then no process can get timestamps lower than ts infinitely many times (this primitive can be implemented in an asynchronous system using read-write registers). The idea of the algorithm is the following. Every process p_i that gets active receives a timestamp in line 4.2 and announces the timestamp in register $T[i]$. Every active process that invokes *try* repeatedly runs a leader election mechanism (lines 4.3–4.6): the non-suspected (by $\Diamond\mathcal{P}$) process that announced the lowest (non-$\perp$) timestamp is elected a leader. If a process p_i is elected, p_i returns from try_i and becomes busy. $\Diamond\mathcal{P}$ guarantees that eventually the same correct active process is elected by all active processes. All other active processes stay blocked until the process resigns and resets its timestamp in line 4.8. The leader executes steps obstruction-free then. Since the leader runs an OF algorithm, the leader eventually resigns and resets its timestamp in line 4.8 so that another active process, which now has the lowest timestamp in T, can become a leader.

Theorem 4. *Algorithm 4 implements a wait-free contention manager.*

Acknowledgements. We are very grateful to Hagit Attiya, Maurice Herlihy, Bastian Pochon, Faith Fich, Victor Luchangco, Mark Moir and Nir Shavit for interesting discussions on the topic of this paper. We would also like to thank the anonymous reviewers for helpful comments.

Algorithm 4: A wait-free contention manager using $\Diamond\mathcal{P}$

```
uses: $T[1,\ldots,N]$—array of registers (other variables are local)
initially: $T[1,\ldots,N] \leftarrow \bot$

4.1 upon $try_i$ do
4.2     if $T[i] = \bot$ then $T[i] \leftarrow GetTimestamp()$
4.3     repeat
4.4         $sact_i \leftarrow \{\, j \mid T[j] \neq \bot \wedge p_j \notin \Diamond\mathcal{P}\text{-}output_i \,\}$
4.5         $leader_i \leftarrow \mathrm{argmin}_{j \in sact_i} T[j]$
4.6     until $leader_i = i$

4.7 upon $resign_i$ do
4.8     $T[i] \leftarrow \bot$
```

References

1. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (1991) 124–149
2. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (1990) 463–492
3. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition). Wiley (2004)
4. LaMarca, A.: A performance evaluation of lock-free synchronization protocols. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94). (1994) 130–140
5. Bershad, B.N.: Practical considerations for non-blocking concurrent objects. In: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems (ICDCS'93). (1993) 264–273
6. Herlihy, M., Luchango, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'93). (2003) 522–529
7. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03). (2003) 92–101
8. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: PODC Workshop on Concurrency and Synchronization in Java Programs. (2004)
9. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05). (2005)
10. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05), LNCS, Springer (2005) 303–323
11. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05). (2005)

12. Fich, F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05). (2005)
13. Guerraoui, R., Herlihy, M., Kapałka, M., Pochon, B.: Robust contention management in software transactional memory. In: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL); in conjunction with the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05). (2005)
14. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43(2)** (1996) 225–267
15. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43(4)** (1996) 685–722
16. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2) (1998) 133–169
17. Guerraoui, R., Kapałka, M., Kouznetsov, P.: Boosting obstruction-freedom with low overhead. Technical report, EPFL (2006) Submitted for publication.
18. Guerraoui, R., Kapałka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Technical report, EPFL (2006)
19. Jayanti, P.: Robust wait-free hierarchies. Journal of the ACM **44**(4) (1997) 592–614
20. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05). (2005)
21. Guerraoui, R., Schiper, A.: "Γ-accurate" failure detectors. In: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96), Springer-Verlag (1996)