Progress Guarantee for Parallel Programs via Bounded Lock-Freedom

Erez Petrank [*]
Computer Science Department
Technion
Haifa 32000
Israel
erez@cs.technion.ac.il

Madanlal Musuvathi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
madanm@microsoft.com

Bjarne Steensgaard
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
Bjarne.Steensgaard@microsoft.com

Abstract

Parallel platforms are becoming ubiquitous with modern computing systems. Many parallel applications attempt to avoid locks in order to achieve high responsiveness, aid scalability, and avoid deadlocks and livelocks. However, avoiding the use of *system locks* does not guarantee that no locks are actually used, because progress inhibitors may occur in subtle ways through various program structures. Notions of progress guarantee such as lock-freedom, wait-freedom, and obstruction-freedom have been proposed in the literature to provide various levels of progress guarantees.

In this paper we formalize the notions of progress guarantees using linear temporal logic (LTL). We concentrate on lock-freedom and propose a variant of it denoted *bounded lock-freedom*, which is more suitable for guaranteeing progress in practical systems. We use this formal definition to build a tool that checks if a concurrent program is bounded lock-free for a given bound. We then study the interaction between programs with progress guarantees and the underlying system (e.g., compilers, runtimes, operating systems, and hardware platforms). We propose a means to argue that an underlying system supports lock-freedom. A composition theorem asserts that bounded lock-free algorithms running on bounded lock-free supporting systems retain bounded lock-freedom for the composed execution.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Parallel programming; D.4.1 [*Software*]: Operating Systems—Synchronization;Concurrency

General Terms Performance, Reliability, Theory, Verification

Keywords Bounded lock-freedom, Lock-freedom, Progress Guarantees, Parallel Computation, Model Checking.

[*] Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06). Part of this work was done while the author was on a sabbatical leave at Microsoft Research.

1. Introduction

Much effort is devoted these days into creating parallel programs that execute efficiently on parallel platforms. Often, attempt is made to avoid using locks, since locks have been known to introduce deadlocks (or livelocks), reduce the responsiveness, and harm the scalability of the application. Traditional synchronization via critical sections guarded by locks are poorly suited for asynchronous systems. If one thread is slow due to a page-fault, a cache-miss, a CPU preemption or even complete failure, all other threads may be consequently delayed. However, the attempt to just avoid employing system locks does not provide the desired application behavior, since finer forms of synchronization (such as compare-and-swap) may block program execution in various subtle ways, even though no specific system lock is held.

Several basic concepts in parallel computation were introduced to capture the essence of progress guarantee. Among them are *lock-freedom* (also known as non-blocking), *wait-freedom*, and *obstruction freedom*. The strongest of these notions is wait-freedom, which requires each thread to make progress whenever it is scheduled for a sufficient number of steps, independently of other concurrently executing threads. Such behavior is indeed desirable, but this requirement is very strong and often results in a complicated and inefficient programs. Lock-freedom requires that when the program threads are run sufficiently long, at least one of the threads make progress. This requirement ensures that the program as a whole makes progress and is never blocked. Lock-free algorithms are known for various tasks and are often efficient. Obstruction freedom is the weakest requirement, stating that progress is only guaranteed if we let one of the program threads run *in isolation* sufficiently long. Typically, threads do not run in isolation on a parallel system. Nevertheless, obstruction freedom is better than no guarantee at all, and if no progress is observed, the system may resort to running isolated threads. In this paper we concentrate on lock-freedom, which seems to be both practicable as well as a desirable guarantee. We also shortly discuss wait- and obstruction-freedom.

Existing definitions for lock-freedom (and other progress guarantees) in the literature only provides guarantees for eventual progress. There is no specific bound on the time to make progress, except that progress time is finite. Furthermore, the (mostly informal) existing definitions are often limited to data structures and not specified for general programs.

The efforts to facilitate non-blocking parallel programming are ubiquitous, appearing for example, in general forms in the support provided by parallel programming libraries for Java [8], or in the efforts put into concurrent non-blocking memory management [2, 17, 24], and in the algorithmic design of parallel algorithms [13].

In this paper we formalize the three notions above using linear temporal logic (LTL) [25]. Next, we propose a definition for *bounded* lock-freedom in which the time for progress is bounded, so that the pace of progress can be clearly determined from the lock-freedom guarantee. It turns out that the details of the formal definition are subtle and simple variations of the definition, that seem intuitively satisfactory, lead to various pitfalls. We discuss those as well.

We use our definition to build an automated tool for checking bounded lock-freedom of concurrent programs. Our tool is based on the CHESS model checker [22] that systematically enumerates all thread interleavings of a concurrent program for a given input harness. We show results from running this tool on the stack data structure, taken from Chapter 11 of [13].

Returning to the design of parallel programs, a common approach in program design in general, is to develop algorithms in a top-down manner. The algorithm is designed at a high level and then implemented on a high-level language. The program is then compiled with a modern compiler, and run on a contemporary operating system and hardware platform. We expect such runs to preserve the good properties of the high level design, and in particular the lock-free guarantee.

However, lock-free algorithms that are implemented on top of modern languages may lose their lock-freedom because the system services that they employ do not support lock-freedom. In some cases, system services may even include explicit or implicit locks. For example, Valois' lock-free linked-list algorithm [26] has a well known C++ implementation by Bush [1], which uses the C++ `new` operation. On most systems, the `new` operation does not maintain lock-freedom. To preserve lock-freedom, a special allocator that preserves lock-freedom must be used. Further examples, discussed in Section 5, show various scenarios in which care is required to maintain lock-freedom through the supporting system and hardware. An interesting question that arises is whether one must explore the entire implementation to the bones of the highly complex hardware in order to make sure that an algorithm satisfies the promised guarantees.

In this paper, we propose a framework that avoids such a need, by separating algorithm design from system support. We formalize and advocate a development framework in which reasoning can be made separately for the program and for the services it uses. Thus, the algorithm developer is able to claim that an algorithm runs in a lock-free manner on *any* system that supports lock-freedom. The system developer is able to reason that the system can support *any* lock-free algorithm. And if both make their claims using our framework, then a simple composition theorem, asserted in this paper, provides the connection, guaranteeing that the run of a lock-free algorithm on a system that supports lock-freedom is lock-free. We stress that the algorithm developer does not need to know the system details and vice versa.

System services are ubiquitous. Examples include runtime services (such as memory management, initialization of program structs, and initialization/termination of threads), interpreter services executing instructions of a program, or micro-instructions that execute machine code, memory hardware that supports caching for memory accesses, virtual memory that supports paging for memory accesses, device drivers that supports access to I/O, profiling code executed with the program to monitor its behavior, etc. All of these must support lock-freedom to ensure that the program running at the highest level is indeed lock-free.

This paper makes the following contributions.

- Formalizing the progress guarantee notions using LTL, in a way that allows formally arguing about progress guarantees of programs.
- Extending the lock-free progress guarantee into the more practical bounded lock-freedom, for which the progress guarantee is bounded for all possible executions.
- Exemplifying the use of the definitions for a verification of the lock-freedom property of a standard lock-free stack implementation.
- Putting up a framework for arguing about system support for lock-freedom, enabling a system designer to show that all lock-free programs will indeed run lock-free on his system.
- Asserting (and proving) a composition theorem that underlies the usefulness of the system support framework.

Organization. In Section 2 we provide basic definitions. We define lock-freedom and discuss variants in Section 3. This definition is used to check an implementation of a lock-free stack using a model checker in Section 4. In Section 5 we demonstrate why a formal approach is advisable in this context by bringing some examples from the real world. A specification of the service framework, i.e., definitions for lock-free programs and lock-free supporting services is given in Section 6. In Section 7 we state and prove the composition theorem. Related work is discussed in Section 8 and we conclude in Section 9.

2. Preliminaries

Lock-freedom is about a program making progress and in particular about an operation execution terminating. To formalize that, we must first establish the terminology for programs, operations, and computation steps.

We will view a *program* as being composed of a series of operations, where *operation* is composed of a sequence of *computation steps* with control statements that direct the flow of execution. We assume that each computation step is executed by a thread belonging to the set T.

Note that the same program can be partitioned into operations in various different ways, according to a choice of granularity. Progress is obtained when an operation terminates. It is possible that a program will be lock-free for one definition of operations and not lock-free with another choice of operation definition. We believe this freedom to choose what progress means is important for proving guarantees that are appropriate for a given program. Thus, operations must be specified and cannot be derived from a program description. In contrast, the computation steps should be automatically deducible from the program. One may choose to think of computation steps of different granularities, such as a Turing Machine δ-transitions, instructions of some machine language, instructions of microcode, or clock ticks of the machine hardware. Terminology should be well set before a proof can be constructed, but for the context of this paper, they should all be equivalent.

To reason about the program execution, we define a set of predicates on computation steps. For a thread $t \in T$, the predicate $sched(t)$ (standing for "schedule of t") is true for a computation step if and only if t is the thread executing the step. Also, the predicate $prog(t)$ (standing for "progress of t") is true for a computation step if and only if t completes an operation, or in other words, makes progress in this step. Obviously, $prog(t) \Rightarrow sched(t)$. Also, $\exists t prog(t)$ is true in a step if some thread makes progress in this step.

An execution of a program P is a sequence of computational steps. Let $P(n)$ be the set of all executions of the program P for any input of size $n \in \mathbb{N}$. With a slight abuse of notation, we represent P to be $\bigcup_{\forall n \in \mathbb{N}} P(n)$, the set of all executions for the program P. Regarding the model of parallelism, in this work we adopt the model of sequentially consistent executions. This allows discussing points in the execution. Extensions for non-sequentially consistent platforms are outside the scope of this paper.

3. Defining Lock-Freedom

We now turn to defining lock-freedom. The formal definitions below are close in spirit to existing definitions in the literature, except that they are specified for general programs.

We use linear temporal logic (LTL) [25] as our formalism. For completeness, we provide a brief introduction to the temporal operators G and F, specialized for our setting. [1] Given an execution $s_0, s_1, s_2, \ldots$, and a predicate p, the predicate $G\ p$ (standing for "globally p") is true at a step s_i if and only if p is true for all steps $s_j, j \geq i$. Also, $F\ p$ (standing for "finally p") is true at a step s_i if and only if p is true for some step $s_j, j \geq i$. A predicate p is true for an execution $s_0, s_1, s_2, \ldots$, if and only if p holds for the first step s_0.

For example, an execution satisfies $Gsched(t)$ if t is the only thread scheduled in this execution. Similarly, $F\ prog(t)$ holds for an execution if the thread t eventually completes an operation in this execution. Finally, $GF\ prog(t)$ is true for an execution if the thread t completes operations "infinitely-often" — in other words, the thread t makes continuous progress in this execution. The definition of lock freedom follows.

Definition 3.1 (Lock-Freedom). *An execution $e \in P$ is lock-free if and only if e satisfies*

$$GF\ \exists t prog(t).$$

A program P is lock-free if every execution $e \in P$ is lock-free.

To reason about bounded progress guarantees, we introduce a temporal operator F_k defined for $k \in \mathbb{N}$ as follows. Given an execution $s_0, s_1, s_2, \ldots$, and a predicate p, we say $F_k\ p$ (standing for "finally p within k") is true at a step s_i if and only if p is true for some step $s_j, i \leq j < i + k$. Intuitively, $F_k\ p$ is true at a step if p holds within the next k steps in this execution. Furthermore, to later define bounded wait-freedom we will need to extend this operator into $F_k^q\ p$ which is true at a step s_i if p holds within the next k steps that satisfy the predicate q.

Bounded lock-freedom is specified with respect to the input length n. This choice will be discussed in Section 3.3 below. We also provide an equivalent non-LTL definition there, in which we spell out the quantifiers and discuss possible alternative definitions.

Definition 3.2 (Bounded Lock-Freedom). *An execution $e \in P$ is k-bounded lock-free if the execution satisfies*

$$GF_k\ \exists t prog(t).$$

A program P is bounded lock-free if for any $n \in \mathbb{N}$, there exists a k such that all executions in $P(n)$ are k-bounded lock-free.

To stress the bound, we sometimes say that P is bounded lock-free with progress guarantee $k(n)$ (and explicitly specify the function $k(n)$).

The traditional (unbounded) lock-free notion nicely captures the theoretical essence of an algorithm that must make progress eventually. A thread does not get stuck forever just because another thread is delayed for a long time or even fails to execute. Theoretically, this is an important guarantee, but the down side is that for such lock-free programs, a delayed (or failing) thread can create a huge delay for the other running threads. It is only guaranteed that it cannot delay them forever. From a practical point of view, and especially when building a supporting service for a practical system, the bounded variant is very attractive since it limits delays even in a worst case scenario. Such a guarantee is essential for systems that must support high responsiveness, and in particular real-time behavior. In the rest of this paper we concentrate on the *bounded* lock-freedom variant. A simple analogue of the definitions, properties, and claims can be also made with the unbounded variant. We chose to work with the bounded definition because of its practical value and since the claims and definitions are slightly more involved to state and prove. It is better to formally cover the more difficult case. In Section 3.3 below, we demonstrate the care required in such definition by showing that slight, seemingly correct variations do not work well.

To complete this section, we also formally define wait-freedom and obstruction-freedom. Wait-freedom ensures that if a thread is scheduled infinitely many times, then it must make progress infinitely many times. Although this is not the focus of this paper, we also provide an extension for bounded wait-freedom (and later for bounded obstruction-freedom).

Definition 3.3 (Wait-Freedom). *An execution $e \in P$ is wait-free if and only if e satisfies*

$$\forall t\,(GF\ sched(t)) \Rightarrow (GF\ prog(t))\,.$$

A program P is wait-free if every execution $e \in P$ is wait-free.

Definition 3.4 (Bounded Wait-Freedom). *An execution $e \in P$ is k-bounded wait-free if the execution satisfies*

$$\forall t\ GF_k^{sched(t)}\ prog(t).$$

A program P is bounded wait-free if for any $n \in \mathbb{N}$, there exists a k such that all executions in $P(n)$ are k-bounded wait-free.

Finally, we formally define obstruction-freedom. This definition makes sure that progress is obtained when a thread runs in isolation.

Definition 3.5 (Obstruction-Freedom). *An execution $e \in P$ is obstruction-free if and only if e satisfies*

$$\forall t GF\ \ (\ sched(t) \Rightarrow prog(t)\ )\,.$$

A program P is obstruction-free if every execution $e \in P$ is obstruction-free.

Definition 3.6 (Bounded Obstruction-Freedom). *An execution $e \in P$ is k-bounded obstruction-free if the execution satisfies*

$$\forall t GF_k\ \ (\ sched(t) \Rightarrow prog(t)\ )\,.$$

A program P is bounded obstruction-free if for any $n \in \mathbb{N}$, there exists a k such that all executions in $P(n)$ are k-bounded obstruction-free.

As explained in the introduction we focus on lock-freedom in the rest of this paper. Wait-freedom is desirable but strong and often hard to achieve for interesting tasks. in contrast, obstruction-freedom provides a guarantee that only holds if a thread is run in isolation. Lock-freedom is practicable and provides a strong enough guarantee.

3.1 Remarks about the Definitions

The wait-free and obstruction-free conditions are local to a thread. This locality may make the requirement simpler to prove. In contrast, proving lock-freedom for a set of threads that create a program, requires a global view of the threads in the set. The proof must take care of the interaction between all threads in the set, and make sure that other threads do not interfere. In particular, it will have to ensure that the threads of the program make progress (when scheduled) *independently* of other threads in the system. The notion of independence will be discussed in Section 3.2 below.

[1] Interested readers should read [25] for a more rigorous and complete treatment.

Definitions 3.2 and 3.1 have been stated with respect to a program. An equivalent definition may be stated for a method, or any other part of the program that consists of full operations[2]. Thus, we can consider also lock-free methods and lock-free sub-programs. It follows trivially that if a program is lock-free then so is any sub-part of it.

Next, we would like to stress that waiting for some event should not be considered a single computation step. A processor that is waiting for a lock to become available or waiting for an event to happen is considered as executing no-operation steps or running a busy wait. If a wait for a lock cannot be bounded, then the algorithm is not lock-free.

Finally, we would like to point out a property that is implied by Definition 3.2 but is not explicitly stated. For most programs bounded lock-freedom implies a bound on the number of threads that can run simultaneously during the program's execution.

OBSERVATION 3.7. *Any non-trivially-progressing bounded lock-free program P and any $n \in \mathbb{N}$, there is a bound $\tau \in \mathbb{N}$ on the number of threads that run concurrently at any point in any execution of P on inputs of length n.*

Non-trivially-progressing programs will be defined below (see Definition 3.8). But their essence becomes clear from the proof sketch of the observation that follows herein. Suppose a program P is lock-free, but for has an input x, such that and an execution on x, for which it can spawn an unbounded number of threads, depending on the execution so far. Let n be the length of x and $k = k(n)$ be the progress guaranteed for inputs of length n. In a way of contradiction, we build a schedule in which the program does not make progress in k collaborative steps, thus foiling the lock-freedom guarantee. The scheduler create an execution in which the program spawns k (or more) threads that run concurrently. It then lets each such thread run until just before making a non-progressing computation step. Typically, not all program steps are progressing, and thus the thread will arrive at executing a non-progressing step. When all threads are about to execute a non-progressing step, the scheduler will let each thread perform one single (non-progressing) step, thus making the program execute k (or more) steps with no progress, contradicting the lock-freedom guarantee.

The above proof is correct only if the scheduler can bring each thread to execute a non-progressing computation step. This holds for standard programs that make progress only once in a while, but we formalize this property in the following definition.

DEFINITION 3.8 (Trivially-progressing programs). *A program P is trivially-progressing if there exists an input x, a state S in the execution of P on x, and a thread of execution T such that if T is scheduled to run exclusively (without any other thread executing any steps) from state S, T makes progress in each and every computation step.*

3.2 Synchronization Independence

Since lock-freedom is about executing cooperatively on a set of threads, then we should be able to also talk about independence of other sets of threads. We thus introduce a simple notion of independent execution. Intuitively, we say that if two programs are lock-free and if they do not communicate in any way, then when they execute together on the same system, each of them is still lock-free.

DEFINITION 3.9 (Synchronization Independence). *Two programs (or sub-programs) are synchronization-independent if for any possible inputs and any possible thread scheduling, the two programs (or sub-programs) never communicate or use shared resources, neither directly, nor via indirect communication with other threads.*

We are now ready to make a simple observation about independence and lock-freedom. The idea is that if two programs do not communicate, then they cannot interfere with each other's progress guarantee. This is true because if one of the programs is allowed to run enough computation steps, then by its original lock-freedom guarantee it must complete an operation, and the second independent program's run does not influence the run of the first program. The formal statement follows.

OBSERVATION 3.10. *Let P_1 and P_2 be two programs that are synchronization independent. If P_1 is lock-free, then running these programs in parallel maintains the lock-freedom of P_1. Namely, P_1's lock-freedom guarantee holds even if P_2 runs in parallel.*

3.3 On defining bounded lock-freedom

In this section we argue that our definition of bounded lock-freedom is the "right" one. As will be shown, slightly modified definitions which may seem adequate do not work out well. The main difference between bounded definition and the standard one is the order of quantifiers. In the bounded version, the bound is guaranteed for all executions (on a given input length) and in the standard one the bound may depend on the execution. To make the discussion more straightforward, we spell out the quantifiers explicitly in the (non-LTL) definition below.

DEFINITION 3.11 (Bounded Lock-Freedom - non LTL version). *A program is bounded lock-free if for any $n \in N$ there exists a natural number k such that, for any possible input x of length n, any possible execution of the program on the input x, and for any possible point t in the execution, there exists an operation whose execution terminates before the program threads collectively execute k steps after time t.*

Definition 3.11 (and its equivalent definition 3.2) specify the guaranteed bound as a function of the input length. We argue that simpler and intuitively correct alternatives fail.

First, consider the simplest definition which guarantees a single constant bound, that should hold for all executions. This definition would first specify an existential quantifier k and demand that all executions must make progress after k collaborate steps. We claim that this definition is too strong, thwarting the ability to prove that any non-trivial algorithm is lock-free. The argument is similar to the one made for Observation 3.7. Consider any multithreaded algorithm that may run an unbounded number of threads simultaneously[3] and in which progress is not trivial, i.e., operations are composed of several steps and do not terminate in each step. Given a bound k on the number of steps required to obtain progress, think of a worst-case in which the program spawns $k + 1$ threads that run concurrently and the scheduler lets each of them execute a single computation step that does not terminate an operation. This execution runs $k + 1$ steps without making any progress, contradicting the k-bounded lock-freedom guarantee. Thus, an independent existential quantifier is too restrictive for modern systems, in spite of its desirable strong guarantee and simplicity.

A seemingly good remedy for the above problem is to make the bound k depend on the number of threads that may run concurrently

[2] We must partition programs on operation boundaries. If we break a program into a sub-program that ends in the middle of an operation, then operation termination cannot be guaranteed when the sub-program finishes executing.

[3] Note that spawning an unbounded number of concurrent threads does not mean spawning an infinite number of threads, which is not realistic. It just means that for each finite number ℓ, there exists an execution that spawns more than ℓ threads.

in the execution. Namely, the definition would say that for any possible τ, there exists a k such that if not more than τ threads are executed concurrently in the execution, then progress within k collaborative steps is guaranteed. Such a definition seems good in its restrictiveness, providing a very strong guarantee. However, since the bound k depends on the number of concurent threads then this definition is too liberal and is not adequate as well. Denote the guarantee $k(\tau)$ to stress the fact that the bound k depends on the bound on the number of concurrently running threads τ. We claim that with this definition any *unbounded* lock-free algorithm becomes also *bounded* lock-free, by showing how to slightly modify any unbounded lock-free algorithm to make it bounded lock-free under this definition. The idea is that when the algorithm encounters an execution that fails to satisfy the guarantee $k(\tau)$, it spawns more concurrent threads (that do nothing) to increase $k(\tau)$ into $k(\tau')$ and buy more time. After spawning enough threads, the desired bound for the specific execution is obtained and the program continues while satisfying the bound $k(\tau')$. This is why we set the guarantee as a function of the input length.

Our chosen definition 3.11 lets the lock-free bound k be determined by the input length n. When proving a lock-freedom property, one will have to show (as part of the proof) a bound $\tau(n)$ on the number of threads as well. Such a bound is required to complete the proof, as our first observation implies, even though our simple definition does not specify it explicitly. A more cumbersome definition could explicitly state the existence of a bound $\tau(n)$ on the number of threads and then a bound $k(\tau)$ on the steps to make progress. (Note that this would fix τ and k for the run, and not allow buying more time by spawning more threads.) Our simpler definition does not allow the bound k to change during the execution, and implicitly achieves the desirable properties.

Sometimes an input is not intuitively associated with a program, e.g., if it is interactive. In such cases, we adopt a convention from computational complexity, and let the program receive an artificial unary input 1^n that sets a complexity bound for its operations and guarantees.

4. Verifying Bounded Lock-Freedom

Given the formal definition of lock-freedom it is possible to build an automated tool for verifying or falsifying lock-freedom of concurrent programs. In this section, we describe a prototype of such a tool built on top of the CHESS [22] model checker. Given a concurrent program and a test harness, CHESS exhaustively enumerates all thread interleavings of the program for the given test harness, verifying safety and liveness properties for the given harness.

Our tool works on the simple observation that for a given bound k, the Definition 3.2 of bounded lock-freedom can be converted into a safety property. Accordingly, we build a finite state monitor, shown in Figure 1, that checks for the violation of k-bounded lock-freedom. Essentially, the monitor "counts" the number of steps since the last progress step and fails if this count becomes greater than k. The model checker performs a synchronous product of this monitor with the state space of the program, and checks if a failure state is reachable.

We chose to check a simple lock-free implementation of the stack data structure, taken from Chapter 11 of [13]. We used an input harness that takes an integer n and creates n concurrent threads, $\lceil \frac{n}{2} \rceil$ of which perform a push operation and the rest peform a pop operation. Each thread terminates after performing its single operation on the data structure. The stack initially contains enough elements for all the pop operations to succeed. Table 1 contains the results of our experiment. We interpreted computation steps as instructions in this scenario. Therefore, we used a monitor that counted the number of instructions executed. We also ran an additional monitor, that counted the number of memory accesses. Such a counter may be valuable for memory-bound applications.

State	:	$s \in [1..k+1]$		
Init	:	$s = 0$		
Transition(t)	:	prog(t)	$\Rightarrow$	$s' = 0$
		!prog(t)	$\Rightarrow$	$s' = s + 1$
				fail if $s' > k$

Figure 1. Finite state monitor that checks for bounded lock-freedom: $GF_k\ \exists t prog(t)$. The state contains a bounded integer s. On a transition of some thread t, the new state s' is set to zero when $prog(t)$ is true, other s' is set to $s + 1$.

# threads	Bound		Time (s)
	# accesses	# instructions	
2	3	17	0.156
3	6	30	0.390
4	9	43	7.800
5	12	56	128.790
6	15	69	2514.238

Table 1. Results for checking bounded lock-freedom for the `LockFreeStack` implementation in Chapter 11 of [13]. We measure bounds in terms of number of shared memory accesses and in the number of instructions.

For each attempted number of threads and for each monitor, we iteratively increased the bound till we found a bound with which the test succeeded. Table 1 reports the time taken for the succeeding bound when run on an Intel Core2 Duo 1.6GHz laptop with 4 GB of memory running a 32-bit Windows Vista operating system. The time roughly reflects the fact that the model checking problem is exponential in the number of threads and linear in the bound.

On looking at the error paths returned by the model checker on failures, we inferred why the memory access bound grows as $3 * (n - 1)$, where n is the number of threads. The push and pop operations essentially perform the following code:

```
while(true){
1:   Node* oldTop = top;
2:   Node* newTop = Compute(oldTop);
3:   if(CompareAndSet(top, oldTop, newTtop))
4:      return;
}
```

Each operation reads the current version of `top`, the variable pointing to the top of the stack and then, depending on the function executed (push or pop), computes the new version of `top`. This computation requires one memory access. Finally, the operation updates the value if `top` did not change during the computation. The `CompareAndSet` operation performs one memory operation on failure and two memory operations on success. For n threads, the maximum length of non-progress computation occurs in the following scenario. All of the n threads have read the same value for `top` are at line 3. One of the thread succeeds. The remaining $n - 1$ threads fail and retry till they reach line 3 again. This takes a total of 3 memory accesses per thread during which none of the threads make progress. Our model checker is currently unable to symbolically prove that $3*(n-1)$ is the lock-freedom bound for n threads. Similar argument can be made for the instruction bound measure.

5. System Support for Lock-Freedom: Motivation

In this section we motivate extending the definitions to handle service support for lock-freedom. We start in Section 5.1 with a simple counter. It is a service that exists in many systems. We show that standard implementations may or may not support lock-freedom, so some care must be used. Next, in Section 5.2 we note that some of the common hardware does not support a worst-cae lock-freedom guarantee. Thus, even when the software is designed and implemented with the proper care, strong guarantees can not be made about progress in the execution. Before going into these examples, we should also mention that the examples of Section 6.1, where the `new` command is used in a lock-free implementation, demonstrates another vulnerability of real lock-free implementations and motivates the formal treatment of lock-freedom support as advocated in this paper.

5.1 An Example: Using Software Event Counters

An important tool used to monitor program behavior for performance improvement or debugging is the performance and event counters service. An event counter service typically provides a set of methods for creating counters and updating counter values. To provide such a service in a multithreaded environment, the service must implement some basic atomic counter access, so that multiple threads can update it consistently. Some systems provide primitives for atomically updating a counter (increment, decrement, and zero its value), while other systems may require use of loops that repeatedly attempt an atomic compare-and-swap (CAS) operation to atomically update the counter. Both cases allow a lock-free implementation of counter updates. In the latter case, when one thread fails to modify the counter atomically, another thread must have succeeded. If τ is a limit on the number of threads that may concurrently execute on inputs of length n, and m is the number of computation steps used to modify the counter in the service implementation, then after the threads execute $m \cdot \tau$ service steps attempting to modify the counter, we know that a program attempt to execute a service operation must have terminated.

The situation becomes more complicated when a background thread is employed. Service support in general may sometimes include a background thread that performs some off-line support. In this case, a background service thread can be used to serve counter values to a separate presentation program (e.g., a graphical performance monitoring tool). As will be discussed formally in the next section, we would like to guarantee that the program's progress is not hindered by the background service thread.

In the counter example, the easy case is when the service thread does not change the counter value. In this case, all races on updating the counter value are due to program threads calling the counter services. Thus, progress must be made by the program on at least one of its threads and the service supports lock-freedom. However, if the service thread may change the counter value, for example to reset the counter value after handling it to the monitoring tool, then lock-freedom support can not be guaranteed. Repeated updates of the counter values by the service background thread may prevent the running program threads from making progress.

Thus, a simple counter service may interfere with the system's progress guarantees. In this paper, we propose a framework for arguing that a service supports lock-freedom.

5.2 Weak LL/SC operations in ISAs

Let us now move to demonstrating the importance of making the system support lock-freedom. In particular, we will claim that widely used systems cannot be used to guarantee support for lock-freedom at this stage. Some substantial hardware and system modifications are required to remedy the problems described below.

Most lock-free algorithms proposed in the literature are expressed either by using CAS operations or by using load-linked and the store-conditional (LL/SC) operations. The LL/SC operations are a pair of instructions often found in reduced instruction set (RISC) instruction set architectures (ISAs). The load-linked (sometimes also called load-with-reservation) instruction retrieves the value of a memory location and simultaneously places a reservation on the memory location. Other threads may modify the memory location, but in doing so they will clear the reservation. The store-conditional instruction stores a value in a memory location if the reservation left by a prior load-conditional instruction has not been cleared. If the write operation is performed, the instruction is said to succeed. If the write operation is not performed, the instruction is said to fail. In its simplest most restricted form, a reservation can be left on at most one memory location (by a single load-linked operation) and the store-conditional operation must operate on the memory location of the prior load-linked instruction.

Many lock-free algorithms have been described in terms of the LL/SC operations. The descriptions generally assume *strong* LL/SC operations, where the store-conditional fails *if and only if* the reserved memory location has been modified. Unfortunately, real hardware does not always implement strong LL/SC operations. For example, the PowerPC and ARM ISAs implement *weak* LL/SC operations for which spurious failures of the store-conditional operations are permitted[4]. For example, reservations will be cleared when performing context switches or when interrupt handlers performs LL/SC operations of their own. Note that interrupt handlers (handling page faults, network communication, etc.) happen at unpredictable times.

Many published lock-free algorithms described in terms of LL/SC operations are not lock-free when the store-conditional operations can spuriously fail. To be lock-free, there must be a bound on the number of steps required to ensure progress, and such bounds can almost always be exceeded by a sufficient number of spurious failures of store-conditional operations, that may be caused by an unfortunate timing of interrupts and context switches. We stress that the event of not making progress is of extremely low probability. In practice, these algorithms will not get stuck. However, a robust worst-case guarantee cannot be claimed. It can only be claimed with high probability.[5]

Thus, the support that these widely available architectures provide for lock-freedom via the implementation of the LL/SC instructions is not good enough to fully guarantee progress in worst-case scenarios for lock-free algorithms. This demonstrates the importance of checking the underlying support when implementing lock-free algorithms on a given system and hardware.

6. Services that Support Lock-Freedom

Let us now consider programs and services. We start with the general definition of a service.

DEFINITION 6.1 (A Service). *A service $\mathcal{S}$ is a support for a set of operations that a caller program may invoke. A service consists of four components.*

[4] Supposedly, the MIPS and Alpha ISAs also only implement weak LL/SC operations, but we have not seen the manufacturers' documentation supporting or disproving this claim.

[5] One may argue that enough disruptions, say, in the forms of interrupts, will foil progress is any case, simply because the program will not get enough CPU cycles to make progress. Indeed lock-free programs cannot make progress if they do not get sufficient CPU cycles. However, a worst-case scenario of spurious failures make lock-free programs fail to make progress even when they do get sufficient CPU cycles.

1. An internal state.
2. A set of operations with a functionality specification for each operation. The functionality specification includes the specification of inputs, the side-effects on the internal state, and the specification of the output as a function of the input and the internal-state.
3. A specification of a valid operation schedule.
4. A background functionality executed concurrently with the caller program execution.

Note that some services do not require all four components in order to provide the service. For example, a mathematical library would typically contain only the second component and empty versions of the other components. A concurrent memory manager will make good use of all four components as follows: The internal state would be the heap location, size, content, and additional allocator information (e.g., a free-list header). The background functionality would consist of the concurrent garbage collector threads. The operations include allocation of an object, and access (read or write) of a field of an object in the heap or local and global variables. A valid schedule of memory management requests will typically allow modifying an object field only after the object has been allocated. It will implicitly require that the object be reachable from the program's local or global variables (the roots), by requiring that the program supplies a pointer to the object. Formally, it is possible to specify the service operations and valid schedule by specifying its operational semantics.

The above definition of a service does not include the actual implementation. Such an implementation may imply more involved interaction between the components, and the actual implementation is what determines the properties of the service. We now separate abstract specification from actual implementation.

DEFINITION 6.2 (An Abstract-Actual Service Specification). *An* abstract-actual *specification of a service consists of two separate specifications of a service.*

1. An abstract specification $\mathcal{A}$ specifies the service operations and their functionality, the valid service operation schedule, and the internal state.
2. An implementation specification $\mathcal{I}$ specifies the entire service including the internal state representation, the steps that implement the functionality of operations, and the steps that implement the background functionality.

We proceed with the definitions that separate reasoning about programs from reasoning about services provided to them. The first definition specifies what should be proven for a program to make sure that it is lock-free in the presence of a lock-free supporting service. We make sure that such a proof is independent of the service implementation, as the same program may run on different systems and the guarantees should be proven once for all possible underlying (supportive) system services.

DEFINITION 6.3 (Program Lock-Freedom). *A program P is lock-free with respect to an abstract service $\mathcal{A}$ if the following two conditions occur.*

1. P is lock-free under the assumption that any service operation is executed immediately, in a single computation step, which produces the functionality specified for the service operation.
2. For any $n \in \mathbb{N}$ there exists a $\tau \in \mathbb{N}$ such that for any input x of length n, and any execution of P on x, there are at most τ concurrent threads at any point in the execution.
3. If the program runs on an input of length n, then it only makes calls to the service operations whose inputs are of length n or less.
4. For any execution of the program P, all its calls to the service operations create a valid service operation schedule.

The first condition in Definition 6.3 formalizes the intuitive manner in which one expects to prove lock-freedom of a program while "ignoring" the underlying system implementation. The service is assumed to happen at almost no cost and to never delay the execution. The second condition explicitly requires a bound on the number of threads that the program runs concurrently. As discussed in Observation 3.7, this is implicitly required for most practical programs, but to make the composition clean, we make this requirement explicit. The third condition ensures that an execution cannot delay a deadline by creating a new service request with a long input. If a service request is made, its input must have a limited size and the response guarantee does not change [6]. This extra condition is not needed in case the bound on the time to make progress is a constant that does not depend on the input.[7] The last condition ensures that the program behaves properly.

We now turn to considering the service and its support of lock-freedom. In order to separate the lock-freedom support from any specification of an actual lock-free program, we conceptually ignore the program operations except for its calls to service operations. This way, we concentrate on the run of any possible valid service requests and ignore the rest of the program operations and the concurrent background threads. Formally, for any execution e of a program P using a service $\mathcal{S}$. we will consider a projection e_S of the execution e on the service operations. This includes all the computation steps that run during the service operation, and excludes all operations of the background threads and operations of the program that are not service requests.

The projection e_S may be thought of as taking a valid program execution and removing all program statements that do not include a service operation call. The service support guarantees should not be concerned with the actual intent of the programs. Therefore, we remove all non-service steps. Note that it is possible to run the operations on the service (given their inputs) without running the entire program, assuming the program only accesses the service's internal state via the service operations.

In the projected execution e_S we care how many steps it takes to finish an execution of a service operation. For service operation steps the predicate *S-prog* is defined to be true if and only if t finishes an execution of a service operation at this step. We call a program *valid* if all its executions make only valid service calls.

DEFINITION 6.4 (Supporting Lock-Freedom). *A projection e_S of an execution e on the operations of Service $\mathcal{S}$ is k-bounded lock-free service execution if e_S satisfies*

$$GF_k\ \exists t\ \textit{S-prog}(t).$$

A service $\mathcal{S}$ is supporting bounded lock-freedom if for any $n \in \mathbb{N}$, and any $\tau \in \mathbb{N}$, there exists a k, such that for any valid program P, and any execution e of P that runs at most τ concurrent threads, and invokes S's operations with inputs of length at most n, the projected execution e_S is a k-bounded lock-free service execution.

Note that typically, the lock-freedom guarantee depends on the number of threads that may be executed concurrently, and hence the dependence of the guarantee k on the number τ of concurrent

[6] the limit n can be made a (reasonably small) function of n, but we chose to keep the definitions simple. The formal input to the program may be artificially padded with 1^n for a large enough n sufficing for all service calls.

[7] Sometimes it is natural to have a constant bound, such as when implementing a stack data structure. At other times, it is important to allow the bound to be dependent on the input of the service request, e.g., when an allocation procedure zeros the space it is allocating.

threads that may be spawned by the calling program. The number τ provides a bound on the number of threads that may run service operations concurrently.

Once the program is proven to be lock-free with respect to an abstract service according to Definition 6.3 above, and the service is proven to support lock-free programs according to Definition 6.4 above, then the composition theorem asserts that the system as a whole is lock-free. The composition theorem is stated and proven in Section 7 below.

We depict the separation of the program from the service in Figures 2-3. Figure 2 depicts the run of a program with service calls and background threads. The program threads are depicted as continuous lines, whose calls to the service operations are depicted as long ovals. The background threads are depicted as dashed lines. Figure 2 shows three program threads and three background threads running concurrently. In Figure 3 the two entities are separated. In 3-a the program part is depicted and the long ovals (representing the service operations) are replaced by a single computation step highlighted by a small circle. Figure 3-b depicts the service part with the background threads and the attainable schedule of service requests obtained from the original execution of the program.

A point worth stressing for the Definition 6.4 in the presence of background threads is that it implicitly requires that enough progress is made by the background threads to guarantee a bounded execution time for each operation execution. The service implementation $\mathcal{I}$ makes such progress while executing the operations or via the background threads. If the service is late because the background threads do not make enough progress, then a service operation may be stalled, and the progress requirement in the definition cannot be guaranteed. Returning to our memory management example, the memory manager may support further allocations only if the garbage collector manages to free space in the background before the heap is exhausted. Therefore, in order to prove that the service supports lock-freedom in this scenario, one may need to assume something about the scheduling of the background threads, ensuring that they get a chance to make sufficient progress. Extensions in this spirit, that allow lock-freedom support to be conditioned on some events happening during the program run, are easy following the definitions provided in this paper.

6.1 An Example: Lock-free Linked-List with Allocation.

We now look at a specific lock-free algorithm from the literature to demonstrate our framework. Several lock-free algorithms for concurrent data structures use helper objects. As a specific example we chose the lock-free linked-list algorithm by Valois [26]. This linked-list algorithm uses auxiliary nodes in between the "real" elements of the list to manage races between deletions and insertions. Details can be found in the original paper, we only highlight relevant issues. In order to execute an insert to the linked list, the abstract insert method is given a new node, a pointer to an object in the list, and an available auxiliary node. But in practice, an implementation of an insert operation must somehow acquire an auxiliary node in order to insert the "real" one into the list. A natural implementation of acquiring an auxiliary node would be to allocate such a node using the system's memory manager, or to manage a pool of available auxiliary nodes for use by the program.

Using our framework to reason about this algorithm, one can show that in its abstract form, this algorithm is lock-free, when assuming that the allocation of an auxiliary node is executed in a single computation step. This is, in fact, the way such algorithms were shown lock-free in the first place. A natural way to define an operation in this context. i.e., define progress, is to think of each operation on the data structure, and in particular an insert operation, as an operation of the program, and its termination be considered progress.

Moving on to the allocation service, a careless implementation of the allocator may use locking to coordinate all allocations in the system, foiling the lock-free guarantee. In particular, the implementation in [1] uses the C++ `new` operation, which on most system will acquire a lock and not maintain lock-freedom. This simple use of allocation while implementing lock-free algorithms is widely seen in the literature, see for example, the two code examples in the Java Concurrency book [8]. To preserve lock-freedom, a lock-free allocator must be used. Another possible implementation of the allocation service is a pool of pre-allocated objects that the allocator can use. This pool of objects can be managed via a lock-free stack. Such a pool does not use background threads, and one can show that when several threads try to allocate an auxiliary object from a non-empty stack, one of them always succeeds prior to executing $k_2 \cdot \tau$ operations, where k_2 is a constant and τ is a bound on the number of threads. Given such a guarantee, the composition theorem guarantees that the entire implementation is lock-free.

Typically, a pool is managed by pre-allocating a number of objects for the pool, and maybe extending it when necessary. A simple implementation will guarantee lock-freedom only if the pool does not need to be extended. This scenario can be phrased as a predicate on program run. Denote by N the number of objects initially allocated for the pool. We define the predicate P on a program execution, to be TRUE if a program execution uses at most N auxiliary objects simultaneously throughout the run. The pooling service supports lock-freedom only for programs that satisfy the predicate P. Using the conditional lock-freedom support terminology, we say that for programs that always satisfies the predicate P, the use of the pooling service is guaranteed to be lock-free.

Having established the framework to reason about lock-freedom separately for programs and services, we are now ready to prove the composition theorem.

7. The Composition Theorem

The theorem that binds the lock-free program guarantee and the lock-free supporting service guarantee into a single lock-free run is the composition theorem stated below. Again, we work with the bounded version of lock-freedom, and everything in this section can be easily translated into the unbounded version.

THEOREM 7.1. *Let P be a program and let $\mathcal{S}$ be a service with an abstraction $\mathcal{A}$ and an implementation $\mathcal{I}$. If the following conditions hold:*

1. *Program P is lock-free with respect to the abstract service $\mathcal{A}$.*
2. *The service $\mathcal{S}$ supports lock-freedom.*
3. *The program P does not communicate with the service $\mathcal{I}$ except via the service operations, and the effects of service calls are exactly and only those effects specified by the abstract service definition.*

Then the joint execution of the program P using the service $\mathcal{S}$ is lock-free.

When we say that execution of the program P using the service $\mathcal{S}$ is lock-free in the assertion above, we mean that after the program runs a limited number of steps, a program operation terminates. In contrast to the abstract step count in Definition 6.3, the counting of program steps in this case includes steps of the program code as well as steps of the service implementation that execute when the program invokes a service request. But not steps executed on the background threads.[8]

[8] To motivate this exclusion, think of concurrent garbage collection. Progress made by the concurrent collector is not considered program's progress.

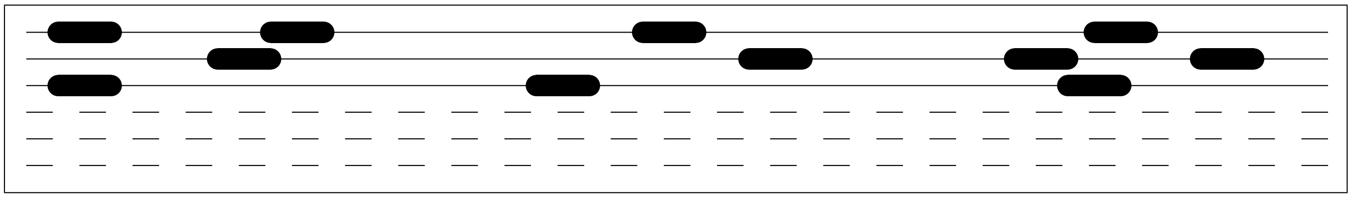
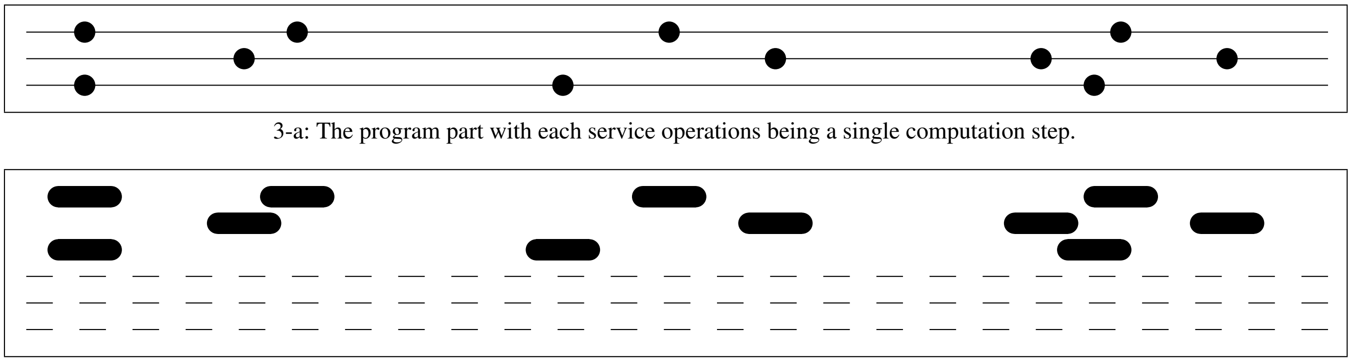


Figure 2. A program running a service. Program threads are represented by continuous lines with service operations depicted as long ovals; service background threads are depicted as dashed lines.



3-a: The program part with each service operations being a single computation step.

3-b: The service part with background threads and service requests.

Figure 3. Our framework separates the program from the service. Figure 3-a depicts the program part and Figure 3-b depicts the service part.

Proof: In order to prove lock-freedom of the system, we need to show that for any $n \in \mathbb{N}$ there exists a k such that for all executions on inputs of length n,

$$GF_k\ \exists t\ prog(t).$$

Fix an input x of length n. By Definition 6.3 Since the program is lock-free with respect to $\mathcal{S}$, it must make calls to the service operations which create a valid service schedule and the input lengths to the service operations are all bounded by n. In addition, it never runs more than τ concurrent threads for some τ that may depend on n. Let k_P be the number of computation steps that guarantee program operation termination as guaranteed by the lock-freedom of the program for inputs of length n, according to Definition 6.3. Let k_S be the number of computation steps that guarantee, according to Definition 6.4 and on inputs of length at most n and at most τ program threads, that a service operation must terminate after a projected execution e_S runs k_S steps. Fix any execution of the program e, and fix any point in the schedule p. We claim that one of the program operations terminates within $k = k_P \cdot k_S$ computation steps executed by the program threads after p.

Since it is guaranteed that the program does not touch the service data structure, except via the service operation invocations, then by Definition 6.4, whenever k_S computation steps of service operations are run, at least one service operation completes, independently of all other steps executed by the program. Thus, after any k_S steps are executed, one event must occur. Either a service step terminates, or a program non-service step is executed. We can therefore deduce that after executing $k = k_S \cdot k_P$ computation steps, at least k_P steps were executed in which either a service request terminated or a non-service step of the was executed. In the abstract scenario of Definition 6.3, we obtain at least k_P steps when service operations are counted as a single step. By the program lock-free guarantee, if at least k_P such steps are executed, then the program makes progress and we are done with the proof of Theorem 7.1. ∎

A similar theorem may be stated for the unbounded variant of lock-freedom. In this case, the proof will first fix the execution of the program with the service; it will then obtain the relevant constants for the program and the service (given the execution) and derive the multiplication, which forms the guaranteed bound for the joint execution.

Wait-freedom satisfies composition in the above sense almost trivially, since each thread makes progress after making k steps, independently of the other threads in the system. Obstruction freedom is also closed under composition, but some subtleties may come up in defining composition properly (similarly to issues that come up with nested atomic transactions)

8. Related Work

Wait-freedom was initially defined by Herlihy [10]: any thread can complete any operation in a finite number of steps, regardless of the execution speeds of the other threads. The term *wait-freedom* is strictly stronger than the term *lock-freedom* studied in this paper. In the same paper, Herlihy [10] also introduced the concept of *nonblocking* implementations. According to that definition some thread is guaranteed to complete an operation in a finite number of steps, regardless of the execution speed of the threads. Massalin introduced the concept of *lock-free* implementations [15], which is synonymous with Herlihy's nonblocking concept. Both terms have since been used to describe various things, but a consensus appears to have emerged in recent years to use the term *lock-free* to describe something close to what Herlihy originally called *nonblocking*.

In a subsequent paper, Herlihy et al. [12] also introduced *obstruction-freedom*. Michael and Scott [21] earlier described this property as lock-free but not non-blocking, however, they used the term "non-blocking" in the sense of "lock-free" and the term "lock-free" to mean without locks. Since Herlihy et al. defined the term "obstruction-free", a concensus appears to have emerged on making a distinction between lock-free and obstruction-free. Prior to the definition of "obstruction-free", the term "lock-free" was sometimes used to describe both categories.

There is a body of work on wait-free implementions of various synchronization operations in terms of other synchronization operations, e.g., Jayanti's implementations of CAS in terms of LL/SC and LL/SC in terms of CAS [14] and Michael's implementation of the latter [19]. Such wait-free "translations" preserve properties of algorithms such as wait-freedom, lock-freedom, or obstruction-freedom.

Many lock-free algorithms were designed and we do not attempt to list them here. Many example appear in [13] Related to the discussion on memory management above, one should note that lock-free supporting algorithms for manually allocating and deleting objects exist and high-performance implementations have been written by Michael [20] and Gidenstam et al. [7].

Our observation that a bound on the number of simultaneous active threads is an important parameter for bounded lock-freedom, has an analogue in a paper by Merritt and Taubenfeld [16], where they study the significance of this bound for several classical distributed algorithms. Other restrictions on the asynchronous model are typically required to obtain interesting algorithms, see, for example [4, 5].

Dongol [3] formalizes (unbounded) progress guarantees using their proposed logic capable of expressing progress. Gotsman et al. [9] have recently built a tool for arguing about (unbounded) progress guarantees using rely-guarantee reasoning.

Automatic reclamation of auxiliary nodes used in various lock-free data structures (such as Valois' lock-free linked-lists) can have a lock-free implementation provided a lock-free mechanism for node allocation and reclamation. Lock-free allocation and reclamation can be achieved by using a concurrently marking garbage collector that supports lock-free allocation, e.g., STOPLESS, CLOVER and CHICKEN [23, 24]. Alternatively, the nodes can be managed by custom mechanisms, e.g., the reference counting scheme by Gidenstam et al. [6], or the pointer guarding schemes of Michael [18] and Herlihy et al. [11], which all limit the number of cursors into a data structure a single thread may maintain. The cost of certain operations are proportional to the number of cursors allowed, so if the limit on the number of cursors is removed, the algorithms will no longer be lock-free.

9. Conclusion

We extended the definition of lock-freedom into bounded lock-freedom, which seems more attractive for applied systems. We formalized the notion of bounded lock-freedom using LTL, incorporated it into the CHESS model checker, and used it to check a simple standard lock-free stack implementation. Next, we advocated a separate reasoning about lock-freedom for a program and the system that supports it. A framework was proposed for separately reasoning about the lock-freedom of a service and about the lock-freedom of fa program. Finally, a composition theorem was shown, asserting that if a program has been shown lock-free according to our framework and the services it uses support lock-freedom, then the real run, given the service implementations is guaranteed to be lock-free as well.

Acknowledgement

We thank Roy Friedman, Tim Harris, Maurice Herlihy, Victor Luchangco, Maged Michael, Mark Moir, Nir Shavit, and David Tarditi for many helpful discussions.

References

[1] Lawrence Bush. Lock free linked list using compare & swap. http://www.cs.rpi.edu/ bushl2/project_web/page5.html, April 2002.

[2] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. *Distributed Computing*, 15:255–271, 2002.

[3] Brijesh Dongol. Formalising progress properties of non-blocking programs. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, pages 284–303, 2006.

[4] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[5] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *DISC*, pages 78–92, 2005.

[6] Anders Gidenstam, Marina Papatriantafilou, Håkon Sundell, and Philippas Tsigas. Practical and efficient lock-free garbage collection based on reference counting. Technical Report 04, Chalmers University of Technology and Göteborg University, 2005.

[7] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. In *Proceedings of the 13th Annual European Symposium on Algorithms*, number 3669 in LNCS, pages 329–342. Springer-Verlag, October 2005.

[8] Brian Goetz, Tim Peierls, Joshua Block, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[9] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 16–28. ACM, 2009.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[11] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamically-sized data structures. *ACM Transactions on Computer Systems*, 23(2):146–196, May 2005.

[12] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[14] Prasad Jayanti. A complete and constant time wait-free implementation of cas from ll/sc and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing*, volume 1499 of *LNCS*, pages 216–230, September 1998.

[15] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[16] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 164–178. Springer-Verlag, 2000.

[17] Maged Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Washington, DC, June 2004. ACM Press.

[18] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.

[19] Maged M. Michael. Practical lock-free and wait-free LL/SC/VL

implementations using 64-bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing*, volume 3274 of *LNCS*, pages 144–158. Springer-Verlag, October 2004.

[20] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

[21] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th International Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[22] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI 08: Operating Systems Design and Implementation*, pages 267–280, 2008.

[23] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. STOPLESS: A real-time garbage collector for multiprocessors. In Mooly Sagiv, editor, *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[24] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of SIGPLAN 2008 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 33–44, Tucson, AZ, June 2008. ACM Press.

[25] Amir Pnueli. The temporal logic of programs. In *FOCS 77: Foundations of Computer Science*, pages 46–57, 1977.

[26] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, Ottawa, Ontario, CA, 1995.