Brief Announcement: Transaction Polymorphism

Vincent Gramoli
EPFL
Switzerland
vincent.gramoli@epfl.ch

Rachid Guerraoui
EPFL
Switzerland
rachid.guerraoui@epfl.ch

ABSTRACT

In this work, we present *transaction polymorphism*, a synchronization technique that provides more control to the programmer than traditional (i.e., *monomorphic*) transactions to achieve comparable performance to generic lock-based and lock-free solutions.

We prove the following results: (i) Lock-based synchronization enables strictly higher concurrency than monomorphic transactions. (ii) Polymorphic transactions enable strictly higher concurrency than monomorphic transactions. The former result indicates that there exist some transactional programs that will never perform as well as their lock-based counterparts, whatever improvement could be made at the hardware level to diminish the overhead associated with transactional accesses. The latter result shows, however, that transaction polymorphism is a promising solution to cope with this issue.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, Concurrent programming structures*; D.3.2 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Algorithms, Theory, Languages, Performance

Keywords

Concurrency, Library

1. TRANSACTIONS FOR EXPERTS

Lock-based and lock-free concurrent implementations of abstract data types are often highly tuned to support a fixed set of efficient features, however, it is difficult to adapt them as they are not generic. For example, a hash table synchronizes efficiently concurrent insert, remove, and contains operations, as long as the number of elements remains proportional to the number of buckets [3]. Unfortunately, this data structure does not support a resize, therefore it is preferable to use a split ordered linked list [4] if one expect the structure to be unbalanced or overloaded.

SPAA'11, June 4–6, 2011, San Jose, California, USA.
ACM 978-1-4503-0743-7/11/06.

The transaction paradigm is an appealing programming idiom for it guarantees to execute in isolation from the other existing transactions. Provided that every operation of an abstract data type is implemented as a transaction, any new operation encapsulated within a transaction will also be atomic. Hence, a novice programmer could reuse such a concurrent library straightforwardly to write other transaction-based concurrent programs. Concurrent programming with transactions is simple in part for this reason and because it consists in delimiting regions of sequential code (e.g., starting with a *start* delimiter). As a drawback, transactions limits concurrency by preventing the programmer from giving hints on the semantics of any transaction. Instead, all transactions execute the same safest semantics—we refer to them as *monomorphic*.

We propose *transaction polymorphism* a novel synchronization technique that allows multiple transactions, with distinct semantics, to run concurrently. To support polymorphism, a transactional memory has simply to accept a semantic parameter p when each transaction starts, e.g., $start(p)$. The application programmer can either set p to the desired semantics or omit it and the default semantics def will be used for the corresponding transaction. Transactional polymorphism has various applications in concurrent programming ranging from providing one liveness guarantee per transaction to distinguishing k-read-modify-write operations from operations whose read-write conflicts do not all impact their linearizability. We illustrate how to use transaction polymorphism to enable greater concurrency.

2. EVALUATING CONCURRENCY

We consider a shared memory partitioned into shared registers, supporting atomic reads/writes, and metadata used for synchronization of set of shared register accesses. A *read* of shared register x that returns value v is denoted by $r(x):v$ or more simply $r(x)$; a write of v on x is denoted by $w(x,v)$ or more simply $w(x)$. An *operation* π is a sequence of read and write accesses to shared registers and a *critical step* γ is a subsequence of an operation.

The *semantics* s of an operation π is an assignment of its accesses to critical steps. For example, the semantics s of a sorted linked list contains operation, $\pi = r(x), r(y), r(z)$, (Figure 1) assigns accesses to two critical steps γ_1 and γ_2 such that $r(x) \mapsto \gamma_1$, $r(y) \mapsto \gamma_1$ and $r(y) \mapsto \gamma_2$, $r(z) \mapsto \gamma_2$ indicating that there should exist a point in the execution where the value returned by $r(x)$ and $r(y)$ where both present, and another point where the values returned by $r(y)$ and $r(z)$ were both present, but not necessarily a point

p_1	p_2	p_3
lock(x)		
$r(x)$		
lock(y)		lock(z)
		$w(z)$
$r(y)$		unlock(z)
unlock(x)		
	lock(x)	
	$w(x)$	
lock(z)	unlock(x)	
$r(z)$		
unlock(y)		
unlock(z)		

p_1	p_2	p_3
start(weak)		
$r(x)$		
		start(def)
		$w(z)$
$r(y)$		commit
	start(def)	
	$w(x)$	
	commit	
$r(z)$		
commit		

Figure 1: Schedule that is accepted by lock-based and polymorphic transactions but not by monomorphic transactions.

at which both values from $r(x)$ and $r(z)$ were present. Intuitively, the semantics of an operation restricts the set of possible schedules comprising its inner accesses by defining its indivisible critical steps.

We consider three operation *synchronizations*: (i) *lock-based synchronization* with $lock(x)$ and $unlock(x)$ functions taking a shared register as a parameter, (ii) *monomorphic synchronization* with $start(\perp)$ and *commit* events delimiting monomorphic transactions, and (iii) *polymorphic synchronization* with $start(p)$ and *commit* events, where p is the semantic hint. A *transactional operation* (resp. *lock-based operation*) is an operation whose set of accesses is extended with the events $start(*)$ and *commit* (resp. $lock(x)$ and $unlock(x)$). (i) A lock-based operation is *well-formed* if for each shared register x every $lock(x)_i$ has a following $unlock(x)_i$ event. (ii) A transactional operation is *well-formed* if it starts with a *start* event and ends by a matching *commit* event. A lock-based (resp. transactional) *schedule* $\mathcal{I}$ is a sequence of events of well-formed lock-based (resp. transactional) operations. Two critical steps γ_1 and γ_2 are concurrent in schedule $\mathcal{I}$ if an event of γ_1 is ordered in $\mathcal{I}$ after the first event of γ_2 but before the last event of γ_2.

Intuitively, a history H is the result of the execution of a schedule $\mathcal{I}$ by synchronization $\mathcal{S}$. More formally, a *transactional history* H_{tx} is the result of the execution of the transactional schedule $\mathcal{I}_{tx}$ by a transactional memory where: (i) $start(*)_i$ events in $\mathcal{I}_{tx}$ are $start(\mathsf{def})_i$ in H_{tx} if $\mathcal{S}$ is the monomorphic synchronization or unchanged otherwise, (ii) one non-*start* event of π_i in $\mathcal{I}_{tx}$ may produce an abort and in this case the schedule is consider *invalid*; and for the remaining events, (iii) for any object x, $r(x)_i$ in $\mathcal{I}_{tx}$ is replaced by its corresponding execution $r(x):v$ in H_{tx} that returns value v, and for any object x, $w(x)$ in $\mathcal{I}_{tx}$ is unchanged in H_{tx}. A *lock-based history* H_ℓ is the result of the execution of the lock-based schedule $\mathcal{I}_\ell$ where for any object x, (i) $r(x)$ in $\mathcal{I}_\ell$ is replaced by its corresponding execution $r(x):v$ that returns value v in H_ℓ, (ii) $w(x)$ and $unlock(x)$ in $\mathcal{I}_\ell$ are unchanged in H_ℓ. Note that the ordering of an input schedule $\mathcal{I}$ is preserved in the resulting history H.

A *sequential history* is a history where no two critical steps are concurrent. A transactional history is *valid* with respect to synchronization $\mathcal{S}$ if it is equivalent to a sequential history and if it does not result from the execution by $\mathcal{S}$ of an invalid schedule (with abort events). (This notion generalizes the input acceptance [2] to $\mathcal{S}$.) A lock-based history is *valid* with respect to synchronization $\mathcal{S}$ if it is equivalent to a sequential history and for each object x, no $lock(x)_i$ occurs between a $lock(x)_j$ and an $unlock(x)_j$ where $i \neq i$.

A schedule is *accepted* by synchronization $\mathcal{S}$ if its execution results in a valid history.

Definition 1 (Concurrency relation). *A synchronization $\mathcal{S}_1$* enables higher concurrency *than synchronization $\mathcal{S}_2$, denoted by $\mathcal{S}_1 \rightrightarrows \mathcal{S}_2$, if there exists a schedule accepted by $\mathcal{S}_1$ that is not accepted by $\mathcal{S}_2$.*

Using this definition, we can strictly compare the concurrency of two synchronizations: $\mathcal{S}_1$ *enables strictly higher concurrency* than another synchronization $\mathcal{S}_2$ if the following properties are satisfied: $\mathcal{S}_1 \rightrightarrows \mathcal{S}_2$ and $\mathcal{S}_2 \not\rightrightarrows \mathcal{S}_1$.

Theorem 1. *Lock-based synchronization enables strictly higher concurrency than monomorphic synchronization.*

The first part of the proof ($\rightrightarrows$) relies on the fact that, unlike lock and unlock events, well-formed transactions are open-close blocks that cannot overlap as depicted by the schedule of Figure 1. The second part ($\not\Leftarrow$) relies on the fact that fine-grained locks can implement 2-phase-locking.

Transaction polymorphism accepts the schedule of Figure 1 by simply using elastic transaction [1] each time a transaction is parameterized with the weak keyword. Next theorem relies on the fact that monomorphic transactions cannot distinguish between semantics $r(x), r(y), r(z) \mapsto \gamma_1$ and semantics $r(x)', r(y)' \mapsto \gamma_{1'}$ and $r(y)', r(z)' \mapsto \gamma_{2'}$ for the operation of p_1 in Figure 1 implying the existence of inconsistent operations or unaccepted schedules.

Theorem 2. *Polymorphic synchronization enables strictly higher concurrency than monomorphic synchronization.*

3. CONCLUDING REMARKS

Transaction polymorphism allows the programmer to control the semantics of transactional operations to avoid concurrency limitations.

Transaction polymorphism raises important questions on the composition of transaction semantics in a common TM implementation. First, how to ensure that two transactions with different semantics could run concurrently without impacting each other semantics? For example, a multi versioned transaction could not return stale data if a singly versioned transaction does not backup data when overwritting it. Second, what should be the semantics of a nested transaction? the semantics indicated by its parameter as if it was not nested, the parent transaction semantics, or the strongest of the two?

4. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and Petr Kuznetsov for their helpful comments. This work is supported in part by FP7 EU projects 216852 and 248465.

5. REFERENCES

[1] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

[2] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1), 2010.

[3] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, 2002.

[4] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3), 2006.