Think about Performance but Do Not Forget about Concurrency

Vincent Gramoli — NICTA and University of Sydney
Petr Kuznetsov — Télécom ParisTech
Srivatsan Ravi — TU Berlin

Predicting the performance of concurrent programs synchronized with a given primitive on a specific chip multiprocessor is nearly impossible. The crux of the problem is that performance is affected by various causes: Two chip multiprocessors may typically offer different cache line sizes, requiring different cache padding to avoid false sharing. The synchronization used to write a concurrent program impacts the number of context switches (e.g., due to classic mutexes), the number of cache invalidations (e.g., due to test-and-set bouncing) and the metadata management overhead (HTM vs. STM). The underlying memory model may be more or less conservative (PSO vs. TSO). In fact, predicting performance requires to be an expert in concurrent algorithms, programming languages and micro-architectures. In this position paper, we argue in favor of studying concurrency rather than performance.

1 The Chip Multiprocessor Model

Consider a system of n threads sharing memory words $m_1, ..., m_k$ by reading and writing them through accesses $r(m_i)$ and $w(m_j, v)$. (We omit access arguments when they are not necessary.) We assume that no two accesses can occur simultaneously. A *sequential program* is a finite-state automaton P with final states reachable from an initial state through a series of transitions corresponding to accesses. Intuitively, a sequential program represents all its possible executions. Branching conditions in the control flow of the program are illustrated by the multiple final states of P_1 and P_2 in Figure 1. (In P_1 a program may terminates in state 3 or reach state 4 if a predicate holds.)

Synchronizations. A synchronization techniques, is a means to prevent data races in concurrent program by protecting memory accesses.

A *lock* is a blocking primitive in that when a thread acquires a lock then other threads trying to acquire the same lock block until the lock gets released.

A read-modify-write (*rmw*) primitive allows to read a memory word, executes a series of accesses before executing a write that is conditional to the state of the previously read memory word. We only consider single-word read-modify-write. The read is often referred to as a *load-link* whereas the conditional write is often referred to as a *store conditional*, however, we will simply use the *compare-and-swap* (CAS) notation that is used on most common architectures. The corresponding synchronization event, denoted by $\mathsf{cas}(m_i, v_1, v_2)$, is a CAS on memory word m_i that changes its value to v_2 and returns true only if its current value is v_1. If the current value is not v_1, then the CAS returns false.

Transactions consist of a compound statement that encapsulates a region of accesses that execute optimistically: the sequence of accesses within a transaction executes in isolation (the stores effect are not visible from other threads point of view) until the transaction commit. A transaction may abort depending on the interleaving of accesses that occur during the execution and the transactional memory algorithm used to implement these transactions, in which case the transaction rolls its changes back and restarts later on.

Concurrent programs. A *concurrent program* is a finite-state automaton C that results from the cartesian product of sequential programs $\langle P_1, ..., P_n \rangle$ executed by threads $p_1, ..., p_n$. Intuitively, a concurrent program C describes the executions that are all possible interleavings of all memory accesses of the sequential programs $P_1, ..., P_n$, so we denote it by $C = P_1 \parallel ... \parallel P_n$. Figure 2 depicts the concurrent program that result from the cartesian product of the sequential programs P_1, P_2 executed by separate threads. Note that all interleavings are represented here, below we discuss how to prune the incorrect interleavings out.

A *synchronized program* S_C^{sync} of a concurrent program C is a concurrent program whose accesses are encapsulated within synchronization events of synchronization technique *sync*. The way concurrent programs are

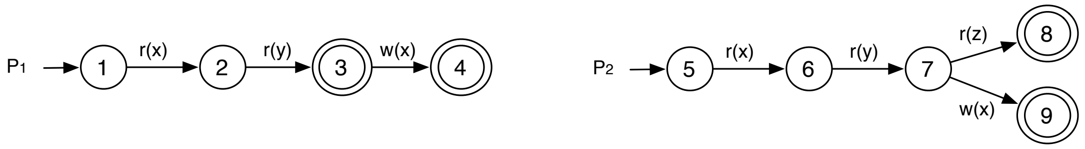

Figure 1: Sequential programs P_1 and P_2 with initial states 1 and 5 and final states $\{3, 4\}$ and $\{8, 9\}$, respectively

synchronized is assumed to be correct, more details can be found in a technical report [6].

Correctness. We define a function *atomic* as a binary relation over shared memory accesses π and π' of a single transaction within an execution α: $atomic(\pi, \pi')$ is true if π and π' appear in α as if they were both occurring at one common indivisible point of the execution. It is important to notice that this relation is not transitive, i.e., $atomic(\pi_1, \pi_2) \wedge atomic(\pi_2, \pi_3) \not\Rightarrow atomic(\pi_1, \pi_3)$. In fact, as π_2 may appear to have executed at several consecutive points of the execution, the points at which π_1 and π_2 appear to have occurred may be disjoint from the points at which π_2 and π_3 appear to have occurred.

This relation can be used to trim out all incorrect schedules from a concurrent program as it gives an information on how and where to use synchronization in the concurrent program. Given a correctness constraint defined by $\forall i$, $atomic(r(x)_i, w(x)_i)$, one has simply to prune out state $\langle 4, 9\rangle$ from C depicted on Figure 2 to obtain a correct concurrent program C'. Informally, this property requires that the value of x read by a thread p should remain unchanged until p writes it. Note that this can be achieved by either acquiring a lock before reading x and releasing it after writing x or using a CAS[1] that takes three arguments: the memory word x, the previously read address v and the argument of the write. However, both would not lead to the same level of concurrency as the lock-based approach would prevent other threads from reading x as well when the lock is held.

Concurrency. As opposed to performance, concurrency can simply be expressed independently from architectural artifacts, making a definition general enough to apply to various cases. Input acceptance [5] measures the variety with which the sequences of accesses of a transactional program can be interleaved, given an ordering of accesses provided by an adversary. Permissiveness [7] measures the variety of histories that can be produced by a transactional program. As opposed to schedules imposed

[1]Note that this assumes that no ABA problem arises.

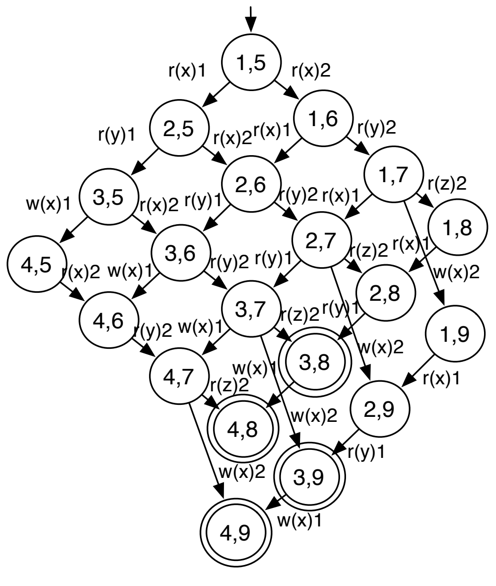

Figure 2: Concurrent programs $C = P_1 \parallel P_2$ with initial state $\langle 1, 5\rangle$ and final states $\{\langle 3, 8\rangle, \langle 3, 9\rangle, \langle 4, 8\rangle, \langle 4, 9\rangle\}$

by an adversary, histories include returned values that can be chosen randomly to maximize permissiveness.

More recently, a new metric of concurrency measures the set of schedules any program can accept, regardless of the way its threads are synchronized [6]. In short, this metric measures the amount of correct schedules of memory accesses that a particular synchronized program accepts without rescheduling it. The fact that the schedule is given by the adversary prevents the synchronization from rescheduling it. This constraint also ensures that the computational effort of the synchronization technique is limited as the synchronization cannot change the schedule without rejecting the original one.

Given such a constraint, the lock-based programs we consider are all pessimistic: lock acquirement and release encapsulate memory accesses originally part of the schedule. This precludes the use of optimistic locking techniques that require validation a posteriori [9]. Appending extra accesses for validation would change the original schedule imposed by the adversary. Transactional and rmw-based synchronized programs that we consider are however optimistic: each non successfully compared-and-swapped or committed sequence of accesses remains invisible from other threads, and their accesses are not considered part of a schedule before becoming visible. This typically precludes any arithmetic errors, like division-by-zero, infinite loop and buffer overflow, that could be due to transient but inconsistent values.

2 Synchronization Techniques

We propose now to investigate the concurrency one can get when using a particular synchronization technique, regardless of the way he can use it. Recall that we simply impose the *atomic* binary relation over the set of accesses of the program.

Drawback of transactions. Transactions are compound statements that encapsulate a series of accesses $\pi_1, ..., \pi_\ell$ that appear to execute atomically. The syntax of these statements forces the transitive closure of the atomic property [3]. More precisely, for accesses π_1, π_2, π_3 executed by the same thread in a program synchronized with transactions we have:

$$atomic(\pi_1, \pi_2) \wedge atomic(\pi_2, \pi_3) \Rightarrow atomic(\pi_1, \pi_3)$$

Note that locks do not have this property. In particular, locks can be held by a single thread at overlapping points of its program so that a thread can acquire a lock on m_2 and release a lock on m_1 that it acquired previously without releasing the lock on m_2. This is precisely the technique used by hand-over-hand locking to guarantee that $atomic(\pi_1, \pi_3)$ is not enforced.

Note that this constraint is known to affect the performance in practice as it prevents transactional programs from accepting simple list-based set schedules [3]: If a synchronized program consists of three threads p_1, p_2 and p_3 each executing $r(x); r(y); r(z)$, $w(x)$ and $w(z)$ within a single transaction, respectively. Figure 3 represents such an existing schedule. We omitted to represent the corresponding concurrent program for presentation simplicity. Note, however, that the concurrent program is the product of the sequential program P_1, P_2 and P_3 that compose the concurrent program have initial states 1, 5 and 7, and final states 4, 5, 8, respectively. This limitation has led to the definition of other transactional models at the expense of

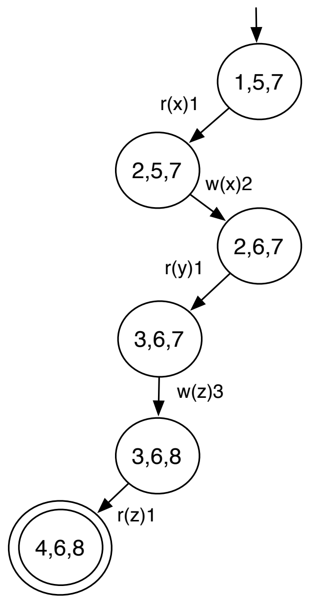

Figure 3: A correct list-based set schedule that is not accepted by classic transactions [3]

breaking the interface, including open nesting [10], transactional boosting [8], elastic transactions [2], view transactions [1], polymorphic transactions [4].

Advantage of optimism. In our definition, executions of programs can be dynamic. This means that the set of accesses are not known prior to execution. In particular, this is illustrated by the branches that are represented in state 7 of Figure 1: in some executions a transition may occur between states 7 and 8 while in others a transition may occur between states 7 and 9. This decision typically depends on some predicate that is evaluated at runtime: this can be, for example, an 'if' statement that checks the value of a node of a linked list. If threads insert and remove nodes concurrently, then the predicate cannot be evaluated statically.

Speculative synchronization likes transactions or rmw-based programs have a significant advantage over pessimistic one in dynamic executions: they can take decision (whether to restart a sequence of accesses) even before this sequence of accesses effectively took place. By contrast, a pessimistic synchronization technique would have to take the decision of protecting some accesses before they occur without knowing what branch will be selected. This often turns out to be an overly conservative decision as pessimistic execution do not take risks.

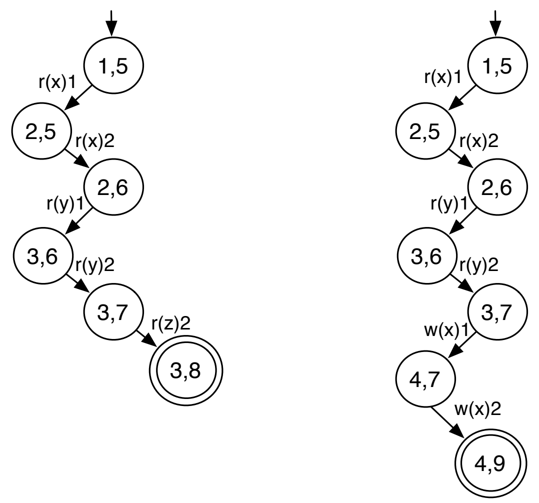

Figure 4: Correct and incorrect schedules that a lock-based program cannot distinguish [6]

This overly conservatism is represented in Figure 4 where a correct schedule (left side) and an incorrect schedule (right side) are indistinguishable from the point of view of a lock-based program until $r(y)1$ returns. Note that the decision of acquiring a lock on x has, however, to be taken before reading y, to guarantee $\forall i, atomic(r(x)_i, w(x)_i)$. Because a lock-based program cannot distinguish these two schedules, in order to be correct it has to take the most conservative decision which corresponds to rejecting these schedules. If it was to accept the correct one, then it would accept the incorrect one as well. This problem arises also in the realistic list-based set scenarios [6] and is easy to remedy with optimistic synchronization as was explained in Section 1.

3 Conclusion

Despite the inherent differences of existing synchronization techniques provided by chip multiprocessors, including read-modify-write, locks, hardware transactional memory, one can reason on the best suited synchronization to maximize the concurrency of an application.

Performance is still the final desirable goal, yet a concurrency metric helps reasoning in terms of potential performance capabilities. While performance of transactions is much higher when they are provided in hardware than in software libraries, their concurrency should remain similar. Concurrency does not capture hardware optimization but rather captures the ability for a program to leverage the shared resources.

As the trend is to increase the number of cores, highly concurrent programs will certainly become more attractive to share cores adequately. In particular, a program that reaches optimal concurrency (without a too large overhead) should intuitively scale to even more cores that those that are currently available on one chip.

References

[1] Y. Afek, A. Morrison, and M. Tzafrir. Brief announcement: View transactions: Transactional model with relaxed consistency checks. In *PODC*, pages 65–66, 2010.

[2] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107, 2009.

[3] V. Gramoli and R. Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, Jan 2014.

[4] V. Gramoli and R. Guerraoui. Reusable concurrent data types. In *ECOOP*, Jul 2014.

[5] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.

[6] V. Gramoli, P. Kuznetsov, and S. Ravi. Optimism for boosting concurrency. Technical Report arXiv:1203.4751v6, arXiv, 2014.

[7] R. Guerraoui, T. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, sep 2008.

[8] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.

[9] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138, 2007.

[10] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, February 2006.