Disjoint-access parallelism does not entail scalability

Rachid Guerraoui and Mihai Letia

EPFL
rachid.guerraoui@epfl.ch, mihai.letia@epfl.ch

Abstract. Disjoint Access Parallelism (DAP) stipulates that operations involving disjoint sets of memory words must be able to progress independently, without interfering with each other. In this work we argue towards revising the two decade old wisdom saying that DAP is a binary condition that splits concurrent programs into scalable and non-scalable. We first present situations where DAP algorithms scale poorly, thus showing that not even algorithms that achieve this property provide scalability under all circumstances. Next, we show that algorithms which violate DAP can sometimes achieve the same scalability and performance as their DAP counterparts. We continue to show how by violating DAP and without sacrificing scalability we are able to circumvent three theoretical results showing that DAP is incompatible with other desirable properties of concurrent programs. Finally we introduce a new property called generalized disjoint-access parallelism (GDAP) which estimates how much of an algorithm is DAP. Algorithms having a large DAP part scale similar to DAP algorithms while not being subject to the same impossibility results.

1 Introduction

As multicores have become the norm, writing concurrent programs that are correct and efficient has become more important than ever. In this context, efficiency is no longer just a matter of making a program fast on a specific number of processors, but also ensuring that when the number of processors is increased, the performance of the program also increases proportionally.

In order to simplify the task of algorithm designers, several attempts to characterize scalable programs have been made. Ideally, these properties would be used in the design phase, when directly measuring scalability is impossible, and still guarantee scalable programs.

One such property is Disjoint Access Parallelism (DAP) [15]. Introduced by Israeli and Rappoport, it has been acclaimed to be both necessary and sufficient for ensuring the scalability of concurrent algorithms. In a nutshell, this property stipulates that operations accessing disjoint sets of memory words must be able to progress independently, without interfering with each other.

Unfortunately, it has been shown to be impossible to achieve DAP along with other desirable properties of concurrent algorithms. Ellen et al. [7] showed for instance that it is impossible to build a disjoint-access parallel universal construction that is wait-free, even when considering a very weak definition of disjoint-access parallelism. To illustrate further, Attiya et al. [4] proved that it is impossible to build a disjoint-access parallel transactional memory having read-only transactions that are invisible and always

terminate successfully, while Guerraoui and Kapalka [9] showed that it is impossible to design a transactional memory that is both disjoint-access parallel and obstruction-free.

Conventional wisdom seems to consider that DAP programs scale under any circumstances while violating this property is catastrophic for scalability. In this work we contradict the two decade old assumption that DAP is necessary and sufficient for obtaining scalable concurrent programs. We first show situations where disjoint-access parallel programs scale poorly, mainly due to the high synchronization cost of specific hardware. We then show how by modifying DAP algorithms in order to violate this property we still obtain good scalability. Surprisingly perhaps, in some cases we find the non-DAP algorithm to outperform a similar DAP one. Although unintuitive, the fact that an algorithm that is not DAP and performs slightly more work can scale better is most likely due to decreasing contention on shared data in a manner similar to flat combining [11].

We use two data structures to evaluate the impact of violating DAP, one lock-based and one lock-free. The lock-based data structure is a closed addressing hashtable that uses lock striping to prevent concurrent threads from accessing the same bucket of the hashtable. The lock-free one is the multi-word compare-and-swap implementation of Harris et al. [10]. In order to observe the effects of losing DAP under several scenarios, we conduct our measurements on two distinct hardware platforms, one being a multi-socket Opteron while the other is a single-socket Niagara.

Using our new findings we revisit three theoretical proofs showing that disjoint-access parallelism is incompatible with other desirable properties of concurrent programs, such as stronger liveness. Then, by circumventing the proofs we show that violating DAP does not hamper scalability or performance, thus making it possible to achieve the other desirable properties without sacrificing scalability.

So far, disjoint-access parallelism has been though of as a binary property, and although in some cases violating it has little to no effect, this is by no means a general principle. To quantify how close the scalability of a non-DAP algorithm is to that of a DAP one, we introduce a new notion called Generalized Disjoint-Access Parallelism (GDAP). In short, GDAP quantifies how much of an operation is DAP.

We experiment with violating the DAP property in two distinct ways. First, by adding a global shared counter we allow restricted communication among processes, for instance allowing one process to observe the presence of another. Then, we allow processes to communicate using a shared queue that permits processes to exchange any type of message. As counter increments feature a lower latency compared to queue operations, the non-DAP part is higher in the latter case, having a more pronounced impact on scalability. Similarly, the latency of hashtable operations is lower than that of the multi-word compare-and-swap, leading to a smaller non-DAP part for the latter. When most of the operation is DAP, even though not all of it, i.e. there is a small non-DAP part, the difference in scalability compared to operations that are fully DAP is negligible and in some cases the GDAP algorithm even achieves better performance and scalability. When a large part of the operation is not DAP, scalability is indeed severely hampered.

The rest of the paper is organized as follows. Section 2 reviews disjoint-access parallelism in a standard model of shared memory. Section 3 describes the benchmarks

we use to show that DAP is neither sufficient (Section 4) nor necessary (Section 5) for ensuring scalability. In Section 6 we review three previous impossibility results relying on DAP and we show that violating this property, under similar scenarios to those in the proofs, has little impact on scalability. We introduce our new notion of generalized disjoint-access parallelism in Section 7 and review related work in Section 8.

2 Disjoint Access Parallelism

We consider a standard model of a shared memory system [5]. Under this model, we first recall the notion of disjoint-access parallelism of Israeli and Rappoport [15].

A finite set of *asynchronous processes* $p_1, \ldots, p_n$ are assumed to apply *primitives* to a set of *base objects* $\mathcal{O}$, located in the shared memory. A primitive that does not change the state of a base object is called a *trivial* primitive. As we wish to reason about the practical performance of disjoint-access parallel programs, we consider base objects to be memory locations supporting operations such as read, write, compare-and-swap, and fetch-and-increment.

A *concurrent object* is a data structure, shared among several processes, implemented using algorithms that apply a set of primitives to underlying base objects, and providing to its user a set of higher-level operations. An *implementation* of concurrent object A from a set of base objects $I \subset \mathcal{O}$ is a set of algorithms, one for each operation of object A. The clients of object A cannot distinguish between A and its implementation.

Two operations affecting distinct concurrent objects are said to be *disjoint-access*. A *transaction* is then defined as a special type of operation that invokes operations of more than one concurrent object. Two transactions are said to be disjoint-access if they access disjoint sets of concurrent objects.

Disjoint-Access Parallelism is a condition on concurrent algorithms stating that any two operations or transactions that access disjoint sets of concurrent objects must not apply primitives to the same base object, but must be able to proceed independently, without interfering with each other. This technique ensures that no hot-spots are created by the implementation and is claimed to ensure scalability by reducing the number of cache misses.

To illustrate, consider a Software Transactional Memory that uses the underlying primitives of the shared memory (read, write, C&S, etc.) to provide the user with read/write registers that can then be accessed through atomic transactions. The registers provided by the STM are then the concurrent objects. In this context, if p_i and p'_i are two processes that execute concurrent transactions T_j and T'_j, DAP requires that if transactions T_j and T'_j access disjoint sets of registers, then they must not access the same base object, i.e. the same underlying memory location. This implies that the time required to execute each of the two transactions would be the same, had they been executing in isolation.

An alternative definition of disjoint-access parallelism allows operations or transactions accessing disjoint sets of concurrent objects to apply trivial primitives to the same base object. Disjoint-access parallelism is only violated if at least one of the primitives is non-trivial. We believe this definition to be more useful in practice as hardware can

typically execute read operations in parallel, while writes are commonly ordered among themselves and with respect to the reads. When arguing that DAP is not a good measure for scalability in practice, we use the latter definition.

3 Benchmarks

We use two different hardware platforms and two separate applications in order to obtain an ample image of the difference DAP makes in the scalability of concurrent programs.

The first platform is a 48-core AMD Opteron equipped with four AMD Opteron 6172 multi-chip modules that contain two 6-core dies each. We further refer to it as the *Opteron*. The L1 contains a 64KiB instruction cache as well as a 64KiB data cache, while the size of the L2 cache is 512KiB. The L3 cache is shared per die and has a total size of 12MiB. The cores are running at 2.1GHz and have access to 128GiB of main memory.

Our other test platform is a Sun Niagara 2, equipped with a single-die SUN UltraSPARC T2 processor. We further refer to it as the *Niagara*. Based on the chip multithreading architecture, this processor contains 8 cores, each able to run a total of 8 hardware threads, totaling 64 threads. The L1 cache is shared among the 8 threads of every core and has a 16KiB instruction cache and 8KiB data cache. The last level cache (LLC) is shared among all the cores and has a size of 4MiB. The cores are running at 1.2GHz and have access to 32GiB of main memory.

Each data point in our graph was obtained by averaging three separate runs. For each run we warm up the JVM for 5 seconds before measuring the throughput for 10 seconds. Due to the long duration of each run, the variation was small enough to be negligible. We continue to describe the two applications we use to assess the degree at which disjoint-access parallelism influences scalability in practice.

3.1 Lock-based hashtable

Our lock-based implementation is based on the striped hashtable of Herlihy and Shavit [13], which in turn is based on the sequential closed-addressing hashtable. Hash conflicts are resolved by assigning elements that map to the same hash value into buckets. Each bucket is protected by a distinct lock and can hold any number of elements by storing them in a linked list.

Although a set implemented using a hashtable cannot be regarded as being DAP due to hash collisions, when considering the hashtable data structure, operations involving the same bucket are no longer logically independent. This allows operations affecting the same bucket to synchronize using the same lock while still satisfying DAP. Operations affecting elements that map to different buckets need to acquire different locks and can proceed independently. The hashtable is the data structure of choice for illustrating DAP in the reference book of Herlihy and Shavit [13].

We made two independent modifications to this data structure in order to violate disjoint-access parallelism. We first added a global shared counter that keeps track of the total number of elements in the hashtable. This counter is incremented by every insert

and decremented by every remove operation of the hashtable using fetch-and-increment and respectively fetch-and-decrement. The hashtable size operation is present in most frameworks for sequential programming, such as that of the JDK. Although approximating the current size of the hashtable can be done by using weak counters, a strong counter is needed in order to provide a linearizable size operation. We thus explore the compromise of losing disjoint-access parallelism in order to obtain an atomic size operation.

The second modification consisted in adding a concurrent queue, shared among all the processes, and making each update to the hashtable also push or pop an element from this queue. While the global counter consists of the minimum violation of DAP, the higher latency of the queue allows us to observe the effects of having a larger part of the operations violate DAP.

3.2 Multi-word compare-and-swap

The multi-word compare-and-swap represents a Java implementation of the algorithm presented by Harris et al. [10]. The algorithm first builds a double-compare single-swap (DCSS) operation out of the compare-and-swap available in hardware and then builds an n-word compare-and-swap operation (NCAS) on top of that. Both the DCSS and NCAS algorithms are based on *descriptors*, making their design non-blocking. Using this mechanism, an operation makes its parameters available so that other processes can provide help in case the initial operation is delayed.

This algorithm is disjoint-access parallel since NCAS operations that affect disjoint sets of memory locations are not required to synchronize among themselves and can proceed in parallel. We again made two independent modifications in order to violate DAP. We first added a global shared counter for keeping track of the number of NCAS operations executed. Although finding this number could have been done by using local counters, we chose this solution in order to obtain a slight violation of disjoint-access parallelism whose impact on scalability we can measure. This solution also allows finding the precise number of NCAS operations executed before the current point in time. The second modification was to make every NCAS operation also perform a push or pop from a concurrent queue, shared among all the processes. Due to the higher latencies incurred by the queue, this modification allows us to test scenarios where operations violate DAP in a larger degree.

4 DAP does not imply scalability

In this section we contradict the common misconception that disjoint-access parallel algorithms necessarily scale. To this aim, we run both the lock-based hashtable and the NCAS algorithms on the Opteron. On this machine the disjoint-access parallel implementations of both algorithms scale poorly. For the sake of comparison, we also plot on the same graphs the versions of these algorithms where DAP is violated by adding a global shared counter.

In Figure 1a we show the scalability of the original (DAP) version of our lock-based hashtable. To put it into perspective, we compare to a version where we break disjoint-access parallelism by having a global counter that stores the size of the data structure.

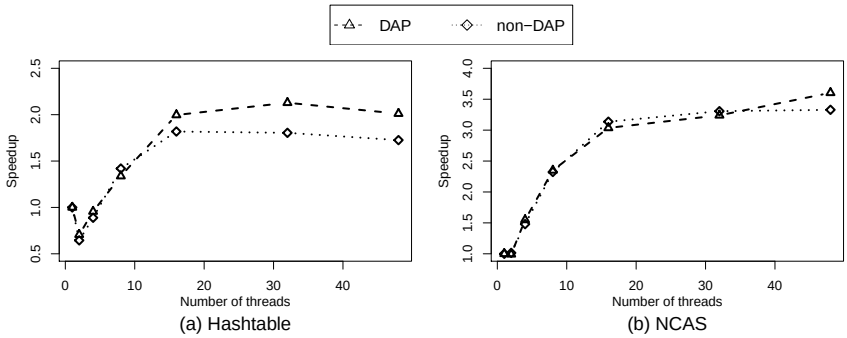

Fig. 1. Speedup obtained when executing 20% update operations on a hashtable with 1024 elements and buckets of length 4 (a) and NCAS operations of length 8 in a system with 1000 locations (b) on the Opteron.

Our experiments use buckets of length 4 and 20% update operations. The DAP version of the hashtable achieves a speedup of only 2.2X on 48 cores compared to the single core performance. The resulting scalability is far from ideal and cannot justify aiming for disjoint-access parallelism when designing a new concurrent algorithm.

In Figure 1b we plot the speedup obtained when running our implementation of the multi-word compare-and-swap on the Opteron. In this experiment each of the NCAS operations attempts to change the value of 8 memory locations, while the system contains 1000 locations in total. In the case of this algorithm, the DAP version achieves a speedup of only 3.5X on 48 cores. To put it into perspective, we also plot the non-DAP version of the NCAS where each operation increments a global shared counter. In this experiment the two algorithms perform almost identically and for some thread counts the non-DAP version performs slightly better. This effect, of having an algorithm that performs strictly more work perform better, is probably caused by decreasing contention on the NCAS locations by using the extra delay provided by the counter. In effect, violating disjoint-access parallelism under this scenario does not bring any performance penalty.

5 Scalability does not imply DAP

In this section we contradict the common misconception that disjoint-access parallelism is a requirement for obtaining scalable concurrent programs. We present experiments using both the lock-based hashtable and the multiword compare-and-swap showing that, for both applications, the non-DAP versions of these algorithms are able to scale. There experiments were conducted on the Niagara machine.

In Figure 2a we plot the speedup obtained when running the hashtable benchmark with 20% update operations on a table with 1024 elements and buckets of length 4. Both the DAP and non-DAP version using a counter scale very well, measuring a speedup of 32X on 64 hardware threads. Both versions scale identically to the point that it is

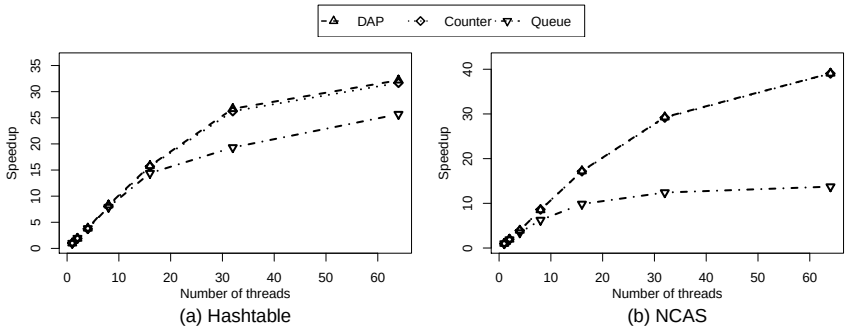

Fig. 2. Speedup obtained when executing 20% update operations on a hashtable with 1024 elements and buckets of length 4 (a) and NCAS operations of length 8 in a system with 1000 locations (b) on the Niagara.

hard to distinguish between the two. The non-DAP version using an additional queue scales less but is still able to reach a 25X speedup on 64 hardware threads. Therefore the small violation of DAP obtained when using an additional counter does not hamper scalability at all, while the larger non-DAP part represented by the queue operations, still allows the algorithm to achieve a 25X speedup.

Figure 2b shows the speedup obtained when executing NCAS operations on our Niagara machine. In these tests, each thread picks 8 locations at random, reads their values using the read operation of the algorithm, and attempts to swap them to a random set of new values. We use a total of 1000 locations for this experiment.

Both the DAP and the non-DAP version using a counter obtain a 40X speedup and, as in the case of the hashtable, their performance is indistinguishable, both versions scaling equally well. The non-DAP version using a queue scales significantly less but is still able to reach a 10X speedup on 64 hardware threads. Compared to the hashtable augmented with the queue, this version of the NCAS scales less due to the fact that all the operations use the queue, whereas in the case of the hashtable, only the updates (20%) were using the queue. Therefore when running our benchmarks on the Niagara machine, disjoint-access-parallelism does not bring any advantage compared to a version of the same algorithm that slightly violates this property by introducing a shared counter. When operations have a larger non-DAP part, such as in the case of adding a shared queue, both the hashtable and the NCAS are able to scale, although not as much as their DAP counterparts.

6 Revisiting impossibilities

In this section we dissect three published theoretical results that we believe are misleading [4, 7, 9]. They seem to indicate that we need to sacrifice liveness in order to have scalability: in fact, we must only sacrifice liveness when aiming for disjoint-access par-

allelism. We put these results to the test by evaluating solutions that circumvent these impossibilities and we show that by weakening DAP, scalability is not affected.

6.1 DAP vs Obstruction-Freedom

The first result [9] proves that it is impossible to design a transactional memory providing transactions that are at the same time disjoint-access parallel and obstruction-free. The latter condition requires that from any point in the execution of the system, if a transaction executes alone for a long enough period of time, it eventually commits. This allows a transaction having a higher priority to be able to preempt or abort lower priority ones at any time and then be sure to commit.

The authors claim that disjoint-access parallelism prevents artificial "hot spots" that may provoke "useless" cache invalidations, thus decreasing performance. We provide experimental measurements showing that even in the case of programs that violate disjoint-access parallelism and feature such artificial "hot spots", the number of cache invalidations does not increase significantly: performance does not suffer.

Circumventing the critical scenario. The authors present the following scenario for showcasing their impossibility result in a system consisting of four transactional variables, x, y, w and z. Transaction T_1 starts executing, reads value 0 for both w and z and then attempts to write value 1 into both x and y. Then T_1 is delayed just before it commits, and T_2 starts executing, reads value 0 from x, writes 1 to w and commits. We observe that T_1 and T_2 cannot both commit since this would violate serializability. Therefore the latter must write a base object to abort the former, which must then be read by a new transaction T_3 that reads y and updates z. Thus, even if T_2 and T_3 access disjoint sets of transactional objects, the latter must read a base object showing that T_1 has been aborted, and that object must in turn have been written by T_2.

One possible way to circumvent the impossibility is to add a counter C_T to every transaction T in the system. In the example above, consider the counter C_{T1} associated with transaction T_1. The counter initially has the value 0 and transaction T_2, before committing, aborts T_1 by incrementing its counter to 1. When T_3 executes, it reads y and also counter C_{T1}, finding that T_1 was aborted.

We estimate a loose upper bound of the performance impact of such a modification by adding a global shared counter to our NCAS system instead of one for each operation. Furthermore, all our NCAS operations increment this global counter instead of only those participating in scenarios similar to those described by the authors. Note that incrementing is at least as expensive as reading the counter value. These two differences have the effect of increasing contention at least as much, if not more than required to contradict the proof.

Performance. In Figure 3 we show the throughput of running our NCAS implementation on a system with 10000 locations. The difference between the DAP and the non-DAP version is that the latter increments a global shared counter on every operation. We vary the length of the NCAS operation between 2 and 64. The results show that when using operations of length at least 8, the two versions of the algorithm perform identically. As we observe shorter lengths of the operations, the difference is small for a

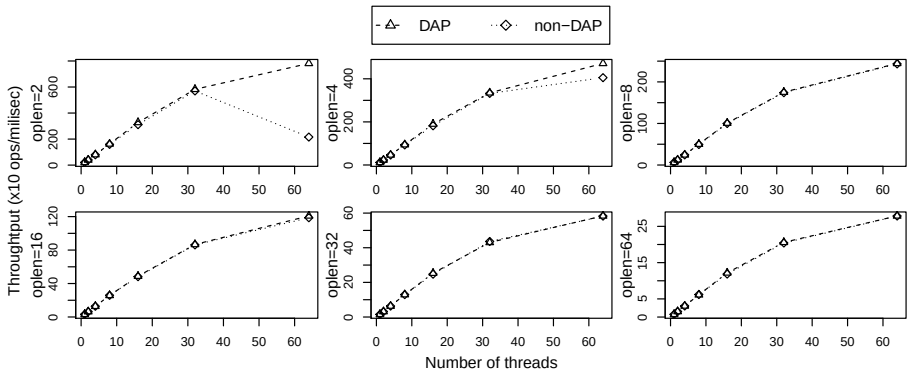

Fig. 3. Throughput obtained when executing NCAS operations of different lengths in a system with 10000 locations on the Niagara.

length of 4 and significant for length 2 but only when running 64 threads. The decrease in performance for the latter case is due to the high contention on the counter caused by the low time required for executing the NCAS. When the NCAS operation needs to write 4 or more locations, contention on the counter decreases and it is no longer a performance bottleneck.

6.2 DAP vs Invisible Reads and Eventual Commit

Attiya et al. [4] showed that it is impossible to build a transactional memory that is disjoint-access parallel and has read-only transactions that are invisible and always eventually commit. They again built on the assumption that disjoint-access parallelism is necessary for achieving scalability. We show however that violating disjoint-access parallelism in a manner that would circumvent their proof has little or no effect on the scalability of the system.

A transactional memory is said to have invisible read-only transactions if such transactions do not apply any updates to base objects; otherwise read-only transactions are said to be visible. Invisible read-only transactions are desirable since this reduces the number of updates to base objects in read-dominated workloads, thus decreasing the number of cache invalidations.

Circumventing the critical scenario. The authors start by defining a flippable execution, consisting of a single long read-only transaction with a complete update transaction interleaved between every two steps of the read-only transaction, such that flipping the order of two consecutive updates is indistinguishable from the initial execution to all the processes. Then they show that in such a flippable execution, the read-only transaction cannot commit. Finally, the authors prove that every disjoint-access parallel transactional memory with invisible read-only transactions has such a flippable execution

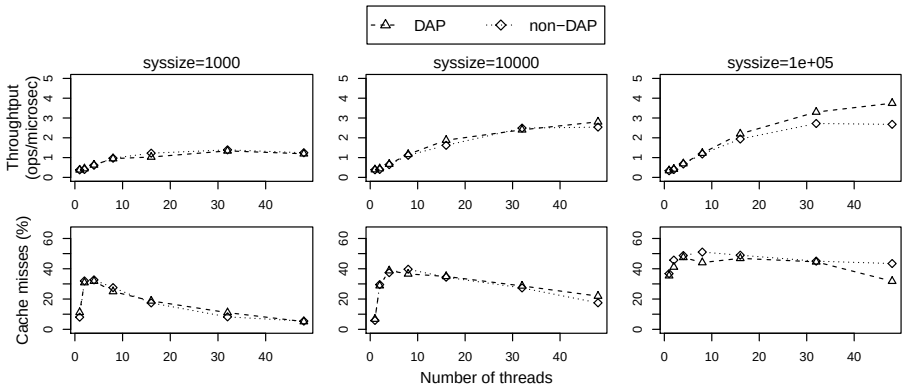

Fig. 4. Throughput and percentage of cache misses obtained when executing NCAS operations of length 8 in a system of different sizes on the Opteron.

and the conclusion follows. The crux of the proof is building an execution where the read-only transaction misses one of two update transactions. Having all transactions increment a shared counter upon committing would enable the read-only transaction to find both update transactions and a flippable execution would no longer be possible.

Performance. In Figure 4 we show both the throughput and the cache miss rate obtained when running the NCAS operations on the Opteron. We use again operations of length 8 and we vary the size of system. The size of the L1 data cache is 64KB, hence systems of 1000 locations fit into the L1. The L2 cache is 512KB, being able to accommodate a system containing 10000 locations. The L3 cache has a total of 12MB and is insufficient to accommodate the largest system size.

One of the main arguments in favor of disjoint-access parallelism is that it increases performance by reducing the number of cache misses in the system. Due to this we perform more in-depth measurements of the cache behavior of the two versions of the NCAS algorithm. We measure the LLC cache miss rate due to its high penalties and because on the Opteron it proves to be a good measure of inter-core communication. We use the perf tool [1], which we attach to our benchmark after performing a 5 second warm-up. To prevent the virtual machine from garbage collecting during our measurements, we use a large initial heap size that is not expected to fill.

For small system sizes we see that both versions of the algorithm do not scale. Due to high contention, operations have a high chance of conflicting, causing them to help each other. As the system size is increased, both algorithms increase in performance but continue to scale poorly. The amount of cache misses is largely the same, with the biggest difference being at 10^5 elements, where a more significant difference in terms of the throughput is observed when reaching 48 cores.

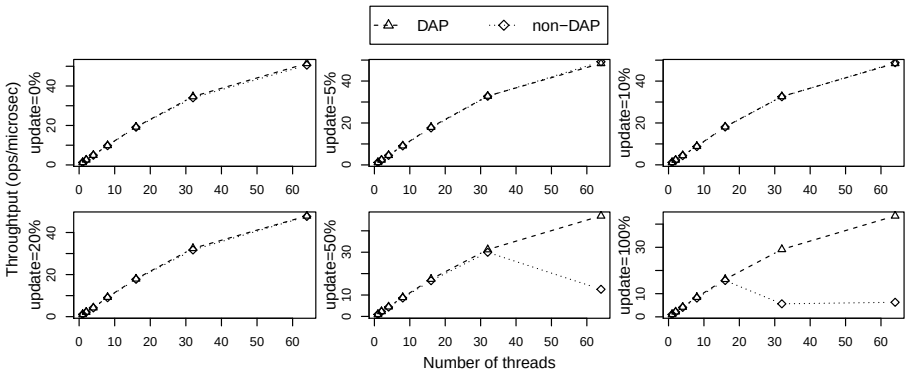

Fig. 5. Throughput obtained when executing different percentages of update operations on a hashtable with 1024 elements and buckets of length 1 on the Niagara.

6.3 DAP vs Wait-Freedom

A universal construction [12] is a concurrent algorithm that takes as input a sequential algorithm and then atomically applies it to a data structure. The main difference between a transactional memory and a universal construction is that the former can complete an operation by returning ABORT, while the latter does not return until it has successfully applied the operation. The universal construction is then equivalent to a transactional memory that reattempts to execute aborted transactions until it succeeds in committing them.

Ellen et al. [7] showed that a universal construction cannot be both disjoint-access parallel and wait-free. Their proof relies on building an unordered linked list with operations append and search. The former adds an element to the end of the list by modifying its tail pointer, while the latter tries to find a specific element by starting from the beginning of the list.

Circumventing the critical scenario. The proof proceeds by having one search operation execute while other processes are continuously executing appends. If the search is not close to the end of the list, it remains disjoint-access with respect to concurrent append operations. However, if the rate at which new elements are appended to the list is faster than the progress of the search operation, the latter will never finish unless the element being searched for is found. It is then sufficient for this element to be different than all the elements being added to the list, and the conclusion follows.

One simple way of circumventing the assumptions in their proof is to allow processes executing a search to read a base object that was written by a process executing append, even though they access disjoint data items. This object could then inform the search that a specific append has a higher timestamp and can be safely be serialized after it.

Performance. In order to evaluate the effect of violating DAP in such a manner, we modified our non-DAP version of the hashtable such that the search operations read the shared counter incremented by the insert and delete. In Figure 5 we compare this new non-DAP version of the hashtable to the original DAP version on the Niagara, while varying the update rate between 0 and 100%. The experiments show that for update rates of up to 20%, the counter does not affect performance at all. Then, when using 50% updates, the effect is visible for thread counts larger than 32, while with 100% updates, the effect becomes visible at 16 threads. As update rates of more than 20% are less common in practice, we conclude that for most workloads adding the extra counter does not affect throughput and scalability.

7 DAP as a non-binary property

So far disjoint-access parallelism has been thought of as a binary property: DAP programs scale, non-DAP programs do not scale. However, in this work we have shown that disjoint-access parallelism is neither necessary (Section 5) nor sufficient (Section 4) for obtaining scalable concurrent programs. To this end we have shown that violating DAP by itself does not make an impact on scalability. Programs that are "almost DAP" scale as well as their fully DAP counterparts.

In order to quantify how close an algorithm is to being disjoint-access parallel, we extend the notion of DAP to define a property called generalized disjoint-access parallelism (GDAP). This property encompasses the classical notion of disjoint-access parallelism but also algorithms that violate it to a certain degree. GDAP is useful for characterizing situations where algorithm designers choose to give up DAP in order to obtain some other desirable properties by capturing how far the resulting algorithm is from its fully DAP counterpart. For instance, many Software Transactional Memories (STMs) use a shared counter [6, 17] in order to achieve faster revalidation or contention management, this way increasing performance.

Intuitively, if an operation OP applies primitives to L base objects, and $1/k$ of these objects are part of a hotspot while the rest is DAP, the only theoretical bound for scalability is when k instances of OP are being executed concurrently. Hence the larger the k factor, the smaller the impact on scalability. In fact, our experiments show that algorithms featuring a sufficiently large k factor still provide the same scalability as fully DAP algorithms and in some cases can outperform them.

Definition 1 (GDAP of order k). *Let $\mathcal{I}$ be the set of implementations of concurrent objects $\mathcal{A}$. If for any two operations OP_i and OP_j such that:*

- *I_i is an implementation of object $A_i \in \mathcal{A}$ and I_j is an implementation of object $A_j \in \mathcal{A}$,*
- *$A_i \neq A_j$,*
- *OP_i is an operation of implementation $I_i \in \mathcal{I}$ and OP_j is an operation of implementation $I_j \in \mathcal{I}$,*
- *I_i applies primitives to base objects O_i and I_j applies primitives to base objects O_j,*
- *$\exists O'_i \subset O_i$ with $(O_i \setminus O'_i) \cap O_j = \emptyset$ and $|O'_i| \times k \leq |O_i|$*

then $\mathcal{I}$ said to be GDAP of order k.

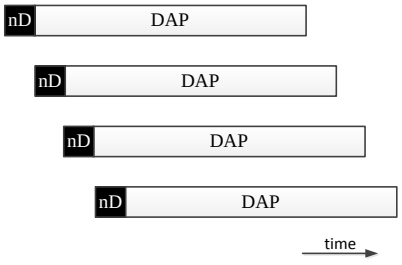

Fig. 6. Execution showing Generalized Disjoint-Access Parallel operations.

Figure 6 shows an execution of four GDAP operations that are not fully DAP. Every operation accesses a common hotspot such that it has a non-DAP part as well as a DAP one. As the non-DAP part is short (large k factor), the four operations can still execute concurrently. This represents the typical scenario resulting from adding a shared object to a set of DAP operations in order to obtain, for instance, a stronger liveness property.

In Figure 3 we observe that operations which are GDAP of a higher order scale better than those of a lower order and can, in fact, perform identically to their fully DAP counterparts. In both experiments we increase the length of the NCAS operations while the non-DAP part remains constant. The result is that longer operations scale better. Switching from length 2 to length 4 provides close to the same performance as the fully DAP version, while for length 8 and greater, the two are practically indistinguishable.

Although in most cases the fully DAP variant of an algorithm scales slightly better than a GDAP variant that is not fully DAP, in certain cases the latter can in fact outperform the former. As shown in Figure 7, the performance gain obtained by adding an extra counter to our hashtable can be as high as 50% for certain thread counts. In this scenario, the threads are executing 10% update operations on a hashtable with buckets of length 1. We show results obtained using tables of size varying from 1024 to 4096 elements. The graphs show that, although the percentage of cache misses stays roughly the same between the DAP and GDAP variants, on some workloads the latter achieves better throughput even though it performs extra work.

8 Related work

As concurrent algorithms become prevalent, knowing what properties to aim for in their design is crucial. In this paper, we contradict classical wisdom by showing that disjoint-access parallelism is neither necessary nor sufficient for ensuring scalability. Instead, we propose a new property, generalized disjoint-access parallelism that helps estimate how close a non-DAP algorithm is to a DAP one. Using this property, algorithm designers can build algorithms that scale similarly to DAP ones but are not subject to the same impossibility results.

The assumption that DAP is sufficient for scalability has been used both when building algorithms that promise good scalability, and as an assumption in proofs. Anderson

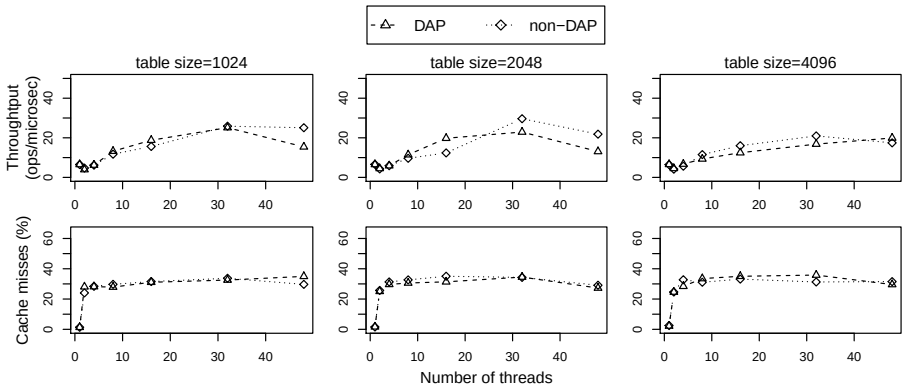

Fig. 7. Throughput and percentage of cache misses obtained when executing 10% update operations on hashtables of different sizes and buckets of length 1 on the Opteron.

and Moir [3] describe universal constructions that are expected to scale well due to being disjoint-access parallel. Kuznetsov and Ravi [16] explore the lower bounds on the number of synchronization operations that a transactional memory must perform in order to guarantee disjoint-access parallelism and the progressiveness liveness condition.

Although DAP seems to be the most accepted theoretical condition for ensuring scalability in practice, other works explore properties that are promising in this direction. Whether or not they are either necessary or sufficient for obtaining scalable concurrent programs remains uncertain.

Afek et al. [2] characterize an operation as having *d-local step complexity* if the number of steps performed by the operation in a given execution interval is bounded by a function of the number of primitives applied within distance d in the conflict graph of the given interval. They define an algorithm as having *d-local contention* if two operations access the same object only if their distance in the conflict graph of their joint execution interval is at most d. Ellen et al. [8] introduce the *obstruction-free step complexity* as the maximum number of steps that an operation needs to perform if no other processes are taking steps.

Imbs and Raynal [14] introduce a property called *help locality* that restricts which other operations can be helped by the current operation. They build upon this property to design an atomic snapshot that scales well, under the assumption that help locality is sufficient for ensuring scalability in practice. However, this assumption has not yet been tested and, similar to disjoint-access parallelism, may have less merit than it receives.

Roy et al. [18] show a tool that profiles concurrent programs giving information about critical sections such as the average time threads spend waiting for a lock and the amount of disjoint-access parallelism that can be exploited. Such a tool can potentially be modified in order to provide the order of GDAP of a concurrent program, helping algorithm designers understand if their scalability issues can be solved by attempting a fully DAP solution.

This paper should be regarded as a step to better understanding scalability. Theoretical conditions that ensure practical scalability are important but, unfortunately, disjoint-access parallelism is not a silver bullet in this regard. As further work, we plan to test other promising theoretical properties in hope to find one that guarantees practical scalability.

References

1. Perf @ONLINE, https://perf.wiki.kernel.org
2. Afek, Y., Merritt, M., Taubenfeld, G., Touitou, D.: Disentangling multi-object operations. In: PODC (1997)
3. Anderson, J.H., Moir, M.: Universal constructions for multi-object operations. In: PODC (1995)
4. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: SPAA (2009)
5. Attiya, H., Jennifer, W.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, 2nd edn. (2004)
6. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI (2009)
7. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal constructions that ensure disjoint-access parallelism and wait-freedom. In: PODC (2012)
8. Fich, F.E., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free step complexity: lock-free DCAS as an example. In: DISC (2005)
9. Guerraoui, R., Kapalka, M.: On obstruction-free transactions. In: SPAA (2008)
10. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Distributed Computing (2002)
11. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA (2010)
12. Herlihy, M.: Wait-free synchronization. TOPLAS (1991)
13. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
14. Imbs, D., Raynal, M.: Help when needed, but no more: Efficient read/write partial snapshot. JPDC (2012)
15. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: PODC (1994)
16. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: OPODIS (2011)
17. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC (2006)
18. Roy, A., Hand, S., Harris, T.: Exploring the limits of disjoint access parallelism. In: HotPar (2009)