The Deterministic Network Calculus Analysis: Reliability Insights and Performance Improvements

Alexander Scheffler, Markus Fögen, Steffen Bondorf
Distributed Computer Systems (DISCO) Lab, TU Kaiserslautern, Germany

***Abstract*—The cost of delay analysis increases fast with size and complexity of a network. Therefore, many recent research efforts in Deterministic Network Calculus (DNC) focused on the tradeoff between accuracy and cost of its delay-bounding analyses. In this paper, we present insights on reliable, reproducible and performance on both branches of DNC, algebraic analysis and optimization, as well as the tools employed by them. We reveal circumstances causing problems for reliability and reproducibility of DNC's optimization analysis and we investigate the potential to improve computational performance of algebraic DNC. To that end, we present theoretical background on the topic of parallelizing the DNC analyses and an implementation in the open-source DiscoDNC tool. With our proposed approach, we achieve a speedup of analysis times of one order of magnitude.**

I. Introduction

Motivation: Deterministic Network Calculus (DNC) is a performance evaluation methodology that derives worst-case bounds on flow delays in communication networks. These bounds are commonly used to verify timing constraints specified for a given network. Deriving them follows a straightforward procedure: first, the network is modeled with the curves defined by the DNC modeling framework and then an analysis derives flow delay bounds. The model must, however, be comprehensive. DNC cannot derive permissible topologies or parameter values. I.e., from a network design perspective, it can only help to rank different alternatives. Therefore, employing DNC in the design phase of a network is restricted to design space exploration [1] where fast runtimes, good scaling behavior and accurate performance bounds are required. DNC promises to scale well as its underlying model – curves bounding the cumulatively resource availability or demand for a duration of observation – is not prone to a state space explosion. Yet, as networks grow larger and more complex, the size and complexity of the DNC analysis computing the desired delay bounds grows, too – in terms of the amount of operations involved in the algebraic DNC analysis (algDNC) [2] or in terms of linear constraints of the optimization-based analysis (optDNC) [3]. This growth of complexity results in long computation times. In this paper, we contribute insights and results on reliability and computational performance of these DNC analyses.

Background: A comparison of complexity and computational effort imposed by modern DNC analyses has been presented in [4]. The authors provide an algebraic analysis that exhaustively executes all permissible sequences of operations to derive a single bound. The most beneficial combination of DNC analysis principles is thus found (see [5] for a comprehensive presentation of analysis principles). A potential combinatorial explosion when combining these principles is mitigated by two strategies: convolution of alternative intermediate arrival curves into a single one and caching of these curves. Implementing both mitigation strategies, algDNC derives competitive delay bounds when compared to optDNC but in considerably shorter computation times. Yet, algDNC execution times can still reach several hours. Thus, we aim to further speed up the analysis to foster its adoption for tasks that require accurate delay bounds in very short times, e.g., data centers and clouds [6], [7]. It was also shown that performance of the optDNC analysis highly depends on the employed optimization software, yet, cannot benefit from parallelization. Potential for parallelization of the algDNC analysis was, however, not investigated before. Work on this topic raises multiple questions addressed in this paper. For example, the central cache may become a bottleneck when parallelizing the algDNC analysis. Figure 1 depicts results of this consideration, revealing that compromising on one of the mitigation strategies as well as a poor choice of parallelization strategy can slow down the analysis considerably.

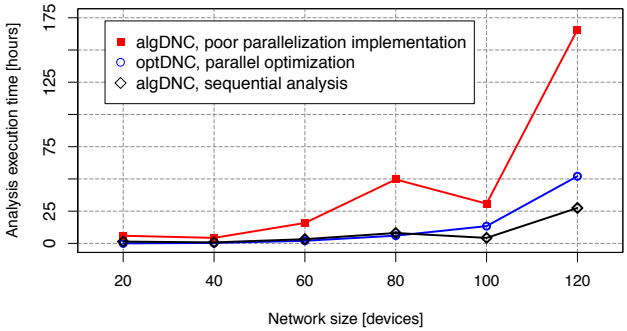


Figure 1: Parallelizing the algebraic network analysis is a non-trivial task that can impose an expensive performance penalty.

Paper Organization: This paper is structured as follows: Section II presents related work. In Section III, reliability and reproducibility of optDNC is investigated. Section IV presents our results on parallelizing the algDNC analysis for modern many-core architectures and Section V concludes the paper.

II. Related Work

The effort imposed by a DNC analysis has been investigated in various ways. Two main sources of computational effort

This work was supported by the Carl Zeiss Foundation.

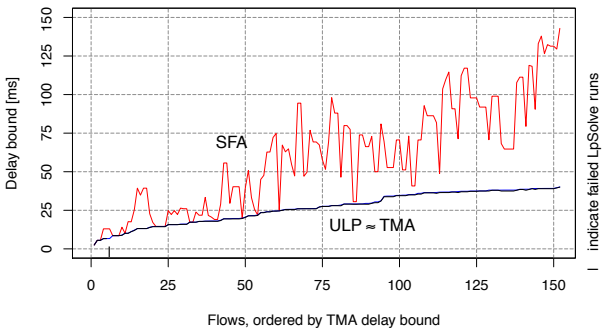
have been addressed in the literature: complexity of the algebraic operations and the complexity of the network analysis. A comprehensive complexity analysis of algebraic operations can be found in [8] and results on the complexity of entire network analysis, i.e., the amount of operations required to accurately bound a flow's end-to-end delay in a network, are provided in [4]. Notable improvements on operations' speed are the theory of compacting curve domains [9], [10], duality of min-plus and max-plus DNC to choose between equivalent operations with different complexities [11], as well as leveraging modern GPGPU technology for the derivation of arrival curves from traces [12] or to perform algDNC operations [13] in parallel. In this paper, we are concerned with the complexity of a network analysis. Work on reducing the (re-)computation effort of an analysis include worst-case model transformations to gain independence of the network topology [14], e.g., based on the worst-case network that is compliant with a specification [15], as well as using different characteristics like the amount of cross-flows multiplexed on a path [16], [17]. Technical mitigation strategies include convolution of arrival curves and caching intermediate results [4]. We contribute the parallelization of the algebraic network analysis to this branch of technical performance improvements.

III. Reliability and Reproducibility

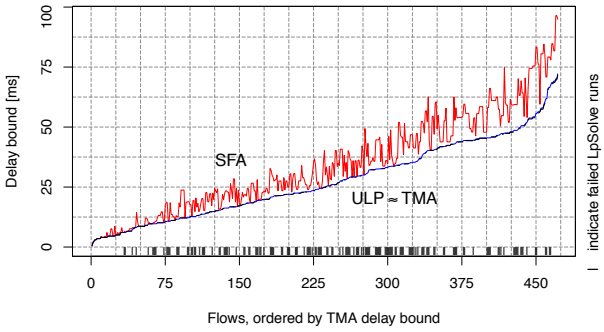
Reliability and reproducibility of results both strongly depend on the tools employed for their computation. Algebraic DNC consists of a sequence of algebraic operations derived from the network. These are applied to the curves that bound worst-case data arrivals and forwarding service. The optimization-based DNC analysis, in contrast, transforms the network to an optimization problem. Such a linear program tends to be large and solving it introduces an inevitable dependency on an LP solver software. In the literature [4], it was shown that the choice of optimization software is decisive for the analysis performance. We extend these findings by new discoveries regarding limited reliability as well as reproducibility. For the evaluation of these aspects, we use the same networks as in [4] and benchmark runs of algDNC's TMA [4] and SFA [2] against optDNC's ULP analysis [3].

A. Reliability

The literature [18], [3] presents optDNC delay bounds derived with the open-source software LpSolve. However, we observed problems with its reliability: one specific flow delay analysis reproducibly failed in the smallest network of our evaluation (20 devices) when employing LpSolve (version 5.5.2.0). Moreover, with increasing network size, the amount of constraints per flow delay LP increases (see Table I) and thus the complexity an LP solver has to cope with grows, too. Again using LpSolve, we observed a rapidly increasing amount of analysis failures in the 40 devices network. Figure 2 shows how the failed flow analyses' delay bounds would have compared to other flow delays in the network – this information was taken from an analysis run with the reliable IBM CPLEX solver (version 12.7.0). In our homogeneous



(a) Network with 20 devices.



(b) Network with 40 devices.

Figure 2: Executing optDNC with LpSolve: failed analyses.

networks, a large delay bound indicates a complex structure of cross-flow entanglement that has to be resolved to a valid scenario upper bounding worst-case interference. Using this proxy-metric for LP complexity, we see that LpSolve fails in a rather simple cross-traffic interference scenario in the 20 devices network (Figure 2a) – the delay bound not computed is in the lower 10th percentile. In the 40 devices network (Figure 2b), failed analyses are scattered over the entire range of complexities (again indicated by delay bounds). Overall, LpSolve not only fails often but also very unpredictably.

B. Reproducibility

We conclude our investigation of optDNC's external tool requirement implications with an important observation regarding reproducibility of results. The ULP analysis is an optDNC heuristic that does not necessarily derive tight bounds in general feed-forward networks. It was derived from the tight LP analysis [3] by reducing the amount of linear programs to a single one that also consists of less constraints than any linear program of the LP analysis. Executing the ULP analysis with

Network size		Constraints per flow	
[devices]	[flows]	[average]	[max]
20	152	3593.70	5258
40	472	54374.25	233435
60	656	201645.26	515272
80	1128	397694.01	1092231
100	1456	665333.80	1981425
120	1592	1514724.68	9005446

Table I: Network sizes and optDNC constraints per flow.

the reliable IBM CPLEX solver, we observed deviations of a single flow's delay bound of up to $0.3‰$. The machine epsilon of double-precision floating-point values is, however, no more than $2^{-53} \approx 2.22 \cdot 10^{-16}$ and should impact results at most once per operation. As the ULP analysis might be tight, such a small deviation can already invalidate the derived bound. Further investigation revealed that the deviations are caused by varying order of constraints in the linear program as well as the solving strategy (i.e., primal simplex or dual simplex). For the smallest network alone, the flow requiring most constraints will have $5258! > 6.5 \cdot 10^{17282}$ permutations, not accounting for permutations of the terms defining each linear constraint. Thus, an exhaustive approach searching for the worst among all permutations' results is not feasible.

Last, note that algDNC does not suffer from this reproducibility issue. We can report reproducible results across the latests major DiscoDNC versions (v2.2.8, v2.3.5, v2.4.0) for equal analyses applied to the networks in Table I. That is, the maximum deviations observed between results of these versions is less than $3 \cdot 10^{-16}$, i.e., well within the expected impact of double-precision floating point rounding errors.

IV. Parallelizing the AlgDNC Network Analysis

This section first presents the theoretical background on parallelizing an algebraic DNC network analysis. We then explore different technical approaches for parallelization and also integrate the caching of intermediate arrival bounds in a parallelized DNC network analysis based on the very recent and very accurate Tandem Matching Analysis (TMA) [4].

A. Potential for Parallelization

Any algDNC analysis starts with the flow whose delay is to be bounded. Its cross-flows are identified and their arrivals are bounded at the locations of interference. To do so, they in turn become the flows to be bounded in a subsequent analysis step. I.e., a recursive procedure is executed that terminates when there is no further cross-flow interference (this termination condition is guaranteed in a feed-forward network). The effort required by this procedure is maximized by the TMA – this algDNC analysis results in the tree structure visualized in Figure 3. Starting at the roots at the top, each intermediate recursion level branches several times in order to test each algDNC analysis principle for superiority over the others. While all computations consist of algebraic operations that can be parallelized as shown in the literature, we aim to parallelize entire sequences of operations. They are depicted as connections between adjacent recursion levels in Figure 3.

The white circles depict synchronization requirements. In terms of a network analysis, they correspond to arrival bounds, i.e., arrival curves for cross-flows at some location in the network. Being arrival curves, all alternatives computed by the branches immediately below the circle can be convolved into a single arrival curve that can also be cached for reuse.

B. Approaches: Technical Details and Evaluation

We present alternative parallelization implementations and evaluate their impact on a multi-core system with a low number of cores but fast single-thread performance (2x Intel Xeon E5420, 8 threads in total) and on a many-core system with a high number of cores but considerably slower single-thread performance (1x Intel Xeon Phi 7210, 256 threads).

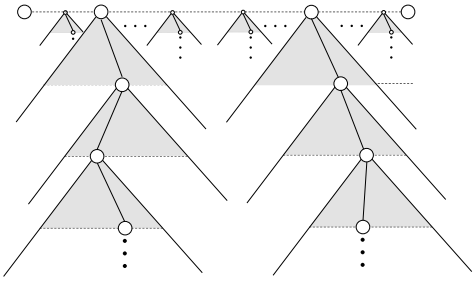


Figure 3: AlgDNC Recursion Tree [4]. (Sub-)Paths from (intermediate) roots to leaves are parallelizable, but caching requires synchronization at every subtree root (white circles).

For the evaluation, a set of example networks with 20 to max. 260 devices from [4] are analyzed. We compute the delay for each flow in each of the networks and measure the overall computation time to bound all flow delays with TMA.

We use the open-source DiscoDNC [19] that is written in Java and thus the parallelization alternatives explored in this section are tied to concepts of this programming language. Yet, most competing closed-source DNC tools are written in Java, too. Most notably, the real-time calculus backend of the MPA Toolbox [20], RTaW Pegase [21], and WOPANets [22] are all written in Java. Thus, our findings are relevant for a wide range of DNC tools.

1) Parallelization of Java 8 Code: Java source code is compiled into bytecode that is then run on a Java Virtual Machine (JVM). In contrast to programming languages that are compiled into machine code, we can leverage the capabilities of the JVM for parallelizing the DiscoDNC network analysis. Alleviating the need to implement comprehensive thread handling in the DiscoDNC itself, two alternatives are available: using the common fork/join pool or separate fork/join pools.

Using the JVM's Common Fork/Join Pool without Caching

Java 8 introduced the concept of parallel streams that allows parallelization of loops with minimal changes to the original code by replacing `for(`element:collection`){...}` with

collection`.parallelStream().forEach(`element`){...}`,

i.e, reformulating independent sets as parallel streams. This approach distributes the operations triggered by elements in the collection over the available threads in the pool. All threads are added to the already existing common fork/join pool of the JVM. Unfortunately, this parallelization technique comes with a drawback. By default, the size of the thread pool equals the number of *available* (virtual) CPU cores – a number that is determined during the start of the JVM. Other applications on the system that were started later may thus be negatively effected and vice versa. To avoid this situation, the dual-Xeon machine (8 threads) only ran the Ubuntu operating system in addition to our experiments. The JVM determined the number

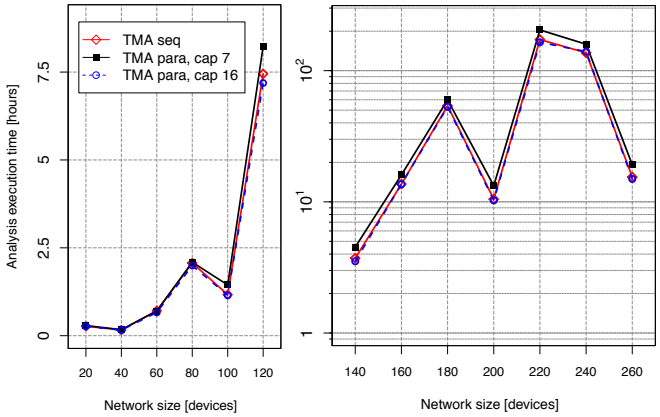


Figure 4: Execution times for sequential mode and common fork/join pool caps of 7 threads (default) and 16 threads.

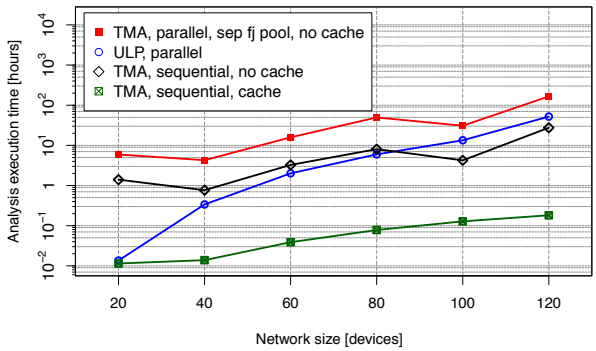


Figure 5: Execution times of the TMA in sequential mode as well as parallelized with separate fork/join pools. Parallel ULP optimization for comparison. Sequential TMA with an arrival bounds cache outperforms any parallelization.

of available CPUs to be 7. However, defining a custom cap of the common thread pool in terms of number of threads is possible. Note that this cannot be done during runtime of a Java program. Instead, the entire JVM must be initialized with

```
-Djava.util.concurrent.
    ForkJoinPool.common.parallelism=n
```

where n denotes the maximum number of threads.

TMA is a compositional feed-forward analysis of DNC that aims to aggregately bound flows in order to obtain a high degree of accuracy. I.e., it consists of two fundamental steps:

1) Given a set of flows, TMA finds the longest tandem of servers these flows cross together. They can be aggregated on this tandem and the flows' available service will be computed using the results of step 2.
2) Arrivals of cross-flows on this tandem are bounded. This step involved defining subsets of cross-flows, according to the servers they cross on the tandem. These flow sets will then be bounded in invocations of step 1.

The entire procedure terminates in the cycle-free feed-forward network when all flows have been traced to their entry point to the network. In addition, TMA exhaustively cuts the tandems into all alternative sequences of subtandems. These are then analyzed individually. Therefore, TMA exhibits the most potential for synchronization among all DNC analyses, filling the common fork/join pool easily. Yet TMA also imposes the highest demand on synchronization of all DNC analyses.

Figure 4 shows the impact of using `parallelStream` on the TMA analysis, run on the dual-Xeon machine where the default number of JVM threads is 7. We observe that this setting is not optimal. The parallelized TMA requires more time than its sequential counterpart; except in the 60 devices network where it is a negligible 34.5 seconds faster. Most notably, the 180 devices network takes an additional 5.95 hours to analyze. Thread handling and synchronization seems to impose a penalty that outweighs parallel execution of subtasks. To further investigate this, we increased the amount of threads in the common pool to 16, hoping that the benefit of parallelization increases faster than the thread handling and synchronization penalty. Indeed, we achieve TMA execution times that are slightly below those of the sequential variant. Interestingly, the 180 devices network that performed worst with less threads in the pool benefitted largely from the increase. The new cap of 16 results in a speed-up of 1 hour.

Next, we investigate a parallelization alternative that vastly increases the amount of threads created by the JVM.

Separate Fork/Join Pools without Caching

Another alternative to avoid the aforementioned search for the best pool size (number of threads) is to use a separate fork/join pool for each `parallelStream`. This is achieved by creating a new `ForkJoinPool` instance with an own `submit`-method to add the respective `parallelStream`. With this approach, we can still set an explicit cap per fork/join pool, yet, we cannot cap the amount of fork/join pools. A large number of pools is to be expected given the branching visualized in Figure 3 and therefore we ran experiments on the 256-thread Xeon Phi machine. A level of parallelism per pool of 1 resulted in more than 7000 threads (i.e., pools). A level of parallelism of 2 caused more than 9000 threads, indicating that only a fraction of pools were filled with two threads. This high number of threads and pools imposes a large management and synchronization overhead on the JVM that is, in contrast to the common fork/join pool, vastly outweighing the benefits of parallelism. Our evaluation results show a seven-fold increase of analysis execution time compared to the sequential computation (see Figures 1 and 5). Thus, we aim at cutting back the need of such expensive synchronization.

C. Creating a Thread-safe and Fast Arrival Bounds Cache

In this Section, we investigate the DiscoDNC's arrival bounds cache that aims to increase the performance of the arrival bounding task by reuse of already computed results. Figure 5 shows that caching outperforms parallelization. Therefore, we aim to create a thread-safe and fast cache in order to benefit from both improvements simultaneously. Yet, this aim imposes the need for a synchronized data structure. Access to

it needs to be synchronized between all threads running the arrival bound calculations in parallel and may thus create a bottleneck slowing down the entire computation. To provide a fast and thread-safe caching solution, we discuss implementation details, cache access as well as synchronization costs and show resulting performance improvements using experimental measurements. The benefit from the cache depends on the cache hits during the arrival bounding tasks. However, because of the size and recursive nature of the arrival bounding this is given (see [4]), especially in large networks.

a) Implementation Details: When the arrival bounds have to be computed for a set of flows, it is checked first if those are already in the cache. To do this in an efficient manner, our advanced cache makes use of hashed maps, i.e., Java's `HashMap` implementation, instead of searching in the set of all entries as the previous proof of concept [4]. In general, hashing can be quite useful when it comes to searching for an entry in a collection since some specific value related to the search term can be used, so only entries with the same value need to be considered, resulting in a significant reduction of the search space. An example would be the server at which the arrival bounds have to be computed for the respective flows we want to bound. Our new arrival bounds cache implementation has two different `HashMap` instances, one holding arrival bounds at servers and one storing arrival bounds for flows crossing a specific link. For brevity, we restrict our presentation to the `HashMap` used for arrival bounds at servers since the other one is similar in nature. We map a server instance to another `HashMap` which, in turn, maps a string to a list of entries that store the arrival bounds. The string contains information about the analysis configuration, among others, the number of the flows to be bounded as well as the multiplexing discipline (e.g., FIFO), in order to further reduce the search space.

Since several threads might be involved in the arrival bounding task, the cache needs to be accessed in a thread-safe way. The main idea is to allow multiple threads to read the respective `HashMap` if it is currently not updated, i.e., several read threads are allowed when there is no thread changing the respective `HashMap`. This has the advantage that read threads do not block each other which should, on average, yield a higher performance compared to using standard mutex locks. On the other hand, one has to consider the possibility that a thread might want to update a `HashMap` while other threads continue getting hits on that map, i.e., just reading it which would mean that the write-thread could be blocked for an arbitrary long time. We use read-write locks to achieve this kind of thread-safety – more precisely, the class `ReentrantReadWriteLock` from the `java.util.concurrent.locks` package – as these locks provide the relevant benefits. The default mode of the framework that we use does not provide a writer preference. Yet, it supports a *fair mode* which uses an approximately arrival-order policy to avoid the issue mentioned above.

Before discussing details of our implementation, we first introduce a notation for the examples that will follow: $w_1 r_0 w_0$ is to be interpreted as the sequence of queries for which the write lock w_0 is older than the query for the read r_0 and write lock w_1, respectively. Suppose that the lock is fully released, i.e. a writer thread released the lock or the last reader thread that had the lock released it. If the longest waiting thread wants to write, it will get the write lock. The following example illustrates this:

$$\ldots\ r_2 r_1 w_2 w_1 r_0 w_0 \rightarrow w_0 \text{ gets the write lock}$$

In the other case, there is a set of threads that want to read and all of those wait longer than the longest waiting write thread. Then, the respective read threads will get the read lock at once:

$$\ldots\ r_4 w_2 w_1 r_3 w_0 r_2 r_1 r_0 \rightarrow \{r_0, r_1, r_2\} \text{ get the read lock}$$

One might ask what happens if some readers currently have the lock and a new read-query occurs: If there is already a waiting write thread then the thread associated with the read-query will not acquire the lock until the oldest currently waiting writer thread got the lock and released it. As a result of this, our implementation ensures that read threads cannot block write-queries for an arbitrary long time.

b) Performance Costs Consideration and Evaluation: The motivation behind the cache was already mentioned above but we also have to consider the inherent additional performance costs. For now, consider only the sequential arrival bounding – thus ignoring synchronization costs. We have costs for the lookup in the cache as well as when updating the cache, i.e., when adding a new entry. Concerning the lookup, we aimed for an efficient search using `HashMap` instances as described above. At first, there will be many updates to the cache since it is almost empty but after some time we will get more and more hits – a simple lookup replaces an entire recursive arrival bounding shown in Figure 3. Thus, the inherent additional performance costs for cache misses are negligible, especially for larger networks. Our evaluation results (shown in Figure 5) clearly show that it is worth caching the arrival bounds, even for a network size of 20 – compare TMA sequential *no cache* and *cache*.

When several threads are involved in the arrival bounding computation, we also have to consider the synchronization costs of accessing the respective `HashMap`. As described above we used read-write locks allowing several read threads to access the data when there is no update queued at this instance of time. Similar as above, at first we have plenty updates, thus mostly write threads, so the parallelism does not fully come into play yet but after the network analysis progressed, the cache hit ratio will increase and the update frequency decreases simultaneously. Having mostly cache hits for some period of analysis execution time means that the pooled threads can work independently since our implementation supports parallel read-access to the cache's `HashMap`.

Our evaluation results already showed that the performance of caching is higher than parallelizing the arrival bound computation alone (without caching). The reason for this is the following: At first we have again a lot of cache misses,

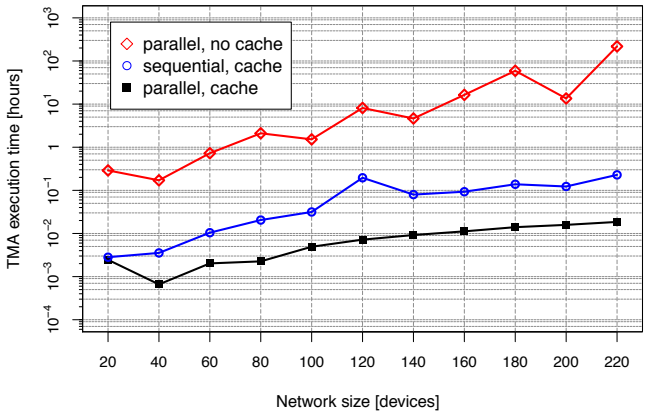


Figure 6: Execution times of the TMA using common-fork/join-pool parallelization, sequential analysis with an arrival bounds cache and our new combination of both.

so the parallel version is expected to be faster (since several threads are involved in the computation) but the cache catches up rather quickly since the cache hits per cache queries will also increase fast due to the recursive nature of the arrival bound computations. After a considerable part of the network has been analyzed, the cached version will mostly just reuse arrival bounds already stored in the cache whereas the previous parallel version still fully computes the bounds (again). As we can see in Figure 6, for the analyzed network of size 100 the parallel version needs more than one hour for the computation (measured on the 256-thread Xeon Phi machine) whereas the sequential cache version is one order of magnitude faster. The deviation between these two versions tends to increase with the network size. E.g., for the network of size 220, the parallel analysis needs more than 100 hours whereas the cached version finishes after less than 1 hour.

Finally, we investigated if the combination of both, caching and parallelism, can further benefit the DNC network analysis performance. Figure 6 shows that this is indeed the case. As mentioned before, we have synchronization costs which decrease fast over time due to an increased cache hit ratio. Hence, after some time the threads can work independently since we mostly have read queries which do not block each other. For example, for the analyzed network of size 140, the computation time using sequential cache is about 6 minutes whereas using the parallel cache just takes about 36 seconds. Overall, for networks > 20 devices, we can observe a performance increase of another order of magnitude with our now thread-safe arrival bounds cache.

V. Conclusion

In this paper, we investigated DNC network analyses regarding reliability and performance. We show that the optimization-based branch of DNC can suffer from reliability and reproducibility issues due to the optimization software that needs to be employed. In contrast, the algebraic DNC analysis is reliable and results are reproducible. Thus, we further improved its performance by parallelizing the analysis procedure. The combination of parallelism and caching of intermediate results turned out to be a non-trivial task. Yet, where other alternatives fail, our implementation of this combination results in a speedup of one order of magnitude.

References

[1] B. Cattelan and S. Bondorf, "Iterative design space exploration for networks requiring performance guarantees," in *Proc. of the 36th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2017.

[2] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[3] A. Bouillard, L. Jouhet, and E. Thierry, "Tight performance bounds in the worst-case analysis of feed-forward networks," in *Proc. of IEEE INFOCOM*, 2010.

[4] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 1, no. 1, pp. 16:1–16:34, 2017.

[5] ——, "Catching corner cases in network calculus – flow segregation can improve accuracy," in *Proc. of GI/ITG MMB*, 2018.

[6] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[7] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. of ACM SIGCOMM*, 2015.

[8] A. Bouillard and E. Thierry, "An algorithmic toolbox for network calculus," *Journal of Discrete Event Dynamic Systems (JDEDS)*, vol. 18, no. 1, pp. 3–49, 2008.

[9] K. Lampka, S. Bondorf, and J. B. Schmitt, "Achieving efficiency without sacrificing model accuracy: Network calculus on compact domains," in *Proc. of IEEE MASCOTS*, 2016.

[10] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan, and W. Yi, "Generalized finitary real-time calculus," in *Proc. of IEEE INFOCOM*, 2017.

[11] J. Liebeherr, "Duality of the max-plus and min-plus network calculus," *Foundations and Trends in Networking*, vol. 11, no. 3-4, pp. 139–282, 2017.

[12] G. Carvajal, M. Salem, N. Benann, and S. Fischmeister, "Enabling rapid construction of arrival curves from execution traces," *IEEE Design & Test*, 2017.

[13] N. Luangsomboon, R. Hesse, and J. Liebeherr, "Fast min-plus convolution and deconvolution on GPUs," in *Proc. of EAI ValueTools*, 2017.

[14] A. Charny and J.-Y. Le Boudec, "Delay bounds in a network with aggregate scheduling," in *Proc. of Quality of Future Internet Services*, 2000.

[15] J. W. Guck, A. Van Bemten, and W. Kellerer, "DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1003–1017, 2017.

[16] I. Chlamtac, A. Faragó, H. Zhang, and A. Fumagalli, "A deterministic approach to the end-to-end analysis of packet flows in connection-oriented networks," *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, pp. 422–431, 1998.

[17] J.-Y. Le Boudec and G. Hébuterne, "Comments on "a deterministic approach to the end-to-end analysis of packet flows in connection oriented networks"," *IEEE/ACM Transactions on Networking*, vol. 8, no. 1, pp. 121–124, 2000.

[18] A. Kiefer, N. Gollan, and J. B. Schmitt, "Searching for tight performance bounds in feed-forward networks," in *Proc. of GI/ITG MMB & DFT*, 2010.

[19] S. Bondorf and J. B. Schmitt, "The DiscoDNC v2 – a comprehensive tool for deterministic network calculus," in *Proc. of EAI ValueTools*, 2014.

[20] E. Wandeler and L. Thiele, "Real-time calculus (RTC) toolbox," `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[21] M. Boyer, J. Migge, and M. Fumey, "PEGASE - a robust and efficient tool for worst-case network traversal time evaluation on AFDX," SAE Technical Paper, 2011.

[22] A. Mifdaoui and H. Ayed, "WOPANets: A tool for worst case performance analysis of embedded networks," in *Proc. of the IEEE CAMAD Workshop*, 2010.