Improving (min,+) convolutions by heuristic operation reordering

Antonio Andrea Salvalaggio, Raffaele Zippo, Giovanni Stea

***Abstract*—Deterministic Network Calculus (DNC) is a mathematical framework for the worst-case analysis of networked systems. In less-than-trivial cases, pen-and-paper computation of DNC expressions is not viable, and the support of a software library for the automated computation is instead required. In particular, the (min,+) convolution operation can be very time-consuming, and several works have focused on optimizing its algorithms. Since the operation is commutative and associative, the convolution of multiple curves can be performed in several ways. In this paper, we compared the runtime performance of different permutations, observing up to an order of magnitude of difference between the worst and best permutations, which highlights an opportunity for optimization. We devised several heuristics based on runtime prediction, to reorder the operations and take advantage of this optimization, obtaining on average an improvement of 30% over the average of all permutations. In this paper, we describe the runtime prediction heuristic approach and its results, highlighting paths for future research.**

***Index Terms*—Deterministic Network Calculus, Worst-Case Analysis, (min,+) Algebra, Performance, Algorithms.**

I. Introduction

Deterministic Network Calculus [1]–[5] and Real-Time Calculus (RTC) [6] are mathematical frameworks for the worst-case analysis of networked systems. While DNC focuses on network traffic, RTC focuses on event-triggered systems. Both frameworks employ characterizations of services guarantees and arrival constraints through cumulative functions of time, which are then composed using operations (min,+) and (max,+) algebra [7], [8]. One such operation is the (min,+) convolution, which is defined as $(f \otimes g)(t) = \inf_{0 \leq s \leq t}\{f(s) + g(t-s)\}$. For example, in DNC, the worst-case service that a packet scheduler guarantees to a flow is characterized by a *service curve* β. If a flow traverses a tandem of schedulers with service curves $\beta_1, \ldots, \beta_n$, we can derive a service curve for the tandem as $\beta_1 \otimes \cdots \otimes \beta_n$, which can be then used to derive worst-case metrics for the tandem. This is called Separated Flow Analysis [5, Section 10.4.2], while a similar property exists in RTC [9]. However, algebraic expressions which may look simple on paper can in fact require lengthy computations [10]–[13]. The research community has therefore developed algorithms [5], [11], [14] and several software packages to automate this task, such as *RTC Toolbox*[15], *RTaW-Pegase*[16], *Nancy*[17], as well as algorithmic optimizations to reduce the time taken to compute each operation, which can be broadly divided in two ways. In the first case, the optimizations exploit properties of the problem that allow for discarding information on the curves without affecting the worst-case metrics computed [18]–[20], while in the second case they exploit novel algebraic properties to compute the same curves at a reduced cost [10], [12], [13].

On the other hand, some phenomena that can affect the runtime cost of a computation cannot be accurately determined before running it. For example, the effect *representation minimization* algorithm [10], which may reduce the size of a curve and thus the cost of chained computations, depends on the shape of the result and cannot be predicted before running the computation. A use case that is affected by this is the computation of chained (min,+) convolutions, such as those used in Separated Flow Analysis. Since the operation is commutative and associative, the result is the same regardless of the order of operations. However, it can be observed that the order does impact the runtime, even if all optimizations of [10], [12] are employed.

In this paper, we 1) investigate the impact of these effects on the runtime using operand permutations, showing that the gap between a "good" and a "bad" permutation is significant, and 2) investigate *heuristics* that can, on average, guide the algorithm towards a "good" permutation.

II. Benchmarking permutations

Since the (min,+) convolution is both commutative and associative, the result of a chained convolution between multiple curves can be computed in many different ways. As discussed in [10], [12], it is possible to greatly reduce computation times of single convolutions between pairs of curves by applying various optimization techniques. The efficacy of these techniques depends on properties of the curves that are being convolved. For this reason, when we compute a chained convolution, the order of computations can affect the efficacy of these optimizations, leading to vastly different computation times.

In this work, we investigated this result exploiting only the commutativity property of (min,+) convolutions.Consider a set of curves $\beta_1, \ldots, \beta_n$. In our experiments, we used $n = 6$. Let π be a permutation of this set, e.g., $\pi = (\beta_5, \beta_3, \beta_2, \beta_6, \beta_4, \beta_1)$. Then we call computing the (min,+) convolution of this set according to π the operation $((((\beta_5 \otimes \beta_3) \otimes \beta_2) \otimes \beta_6) \otimes \beta_4) \otimes \beta_1$. Note that these are $n-1$ convolutions. In these experiments we used sets of six curves, which results in 360 different ways of computing a convolution according to different permutations. We measured and compared the runtimes for each of these permutations. The curves where generated as staircase curves with random step

All authors are with University of Pisa, Department of Information Engineering; A. A. Salvalaggio is also with Scuola Superiore Sant'Anna. e-mail: a.salvalaggio@studenti.unipi.it, raffaele.zippo@ing.unipi.it, giovanni.stea@unipi.it

lengths and heights. We repeated the experiments on seven different sets, generated with different rng seeds. To take accurate measurements, we repeated each computation four times, ignoring the first one (used as a warm-up) and taking the median value between the other three measurements.

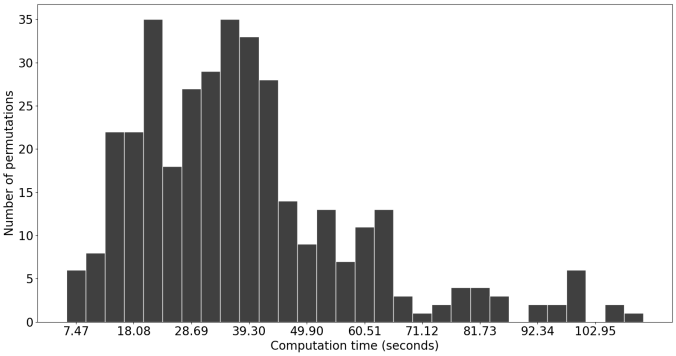

Fig. 1. Distribution of runtimes to compute the (min,+) convolution of a set of 6 curves, according to all possible permutations of the set. Minimum value: 5.7s; Maximum value: 111.8s; Average: 38.48s.

The distribution of computation times across all possible permutations, for one such set of curves, is shown in Figure 1. This result is of similar shape for all the seven sets of curves we analyzed: roughly a left-leaning bell distribution with a long right tail. The difference in runtime between the best and worst permutations is not negligible: in all our experiments (using different sets of curves) the worst ordering takes between five and twenty times longer to compute than the best one. In the case shown in Figure 1, this is the difference between 111.8 and 5.7 seconds.

The left-leaning shape of the distribution suggests that, with a random ordering of operations, one is likely to obtain a runtime significantly lower than the worst one. However, the difference of an order of magnitude indicates that using an ordering algorithm, rather than leaving the ordering to chance, could easily reduce average computation times just by avoiding the long right tail of the distribution. Thus, we set on to search an algorithm able, with low enough overhead, to find a "good ordering", i.e., one that excludes the right tail of the distribution.

We envisioned our algorithm as an iterative one that reorders the curves as it runs computations. In the following, we use $\mathcal{C}$ to indicate the set of curves to convolve, β_i to indicate a curve from $\mathcal{C}$, and β_j^r to indicate the intermediate result of the j-th convolution. The iterative algorithm would behave as follows.

- At the first step, it selects the first two curves to convolve. Let them be β_{i_1} and β_{i_2}, it computes $\beta_1^r = \beta_{i_1} \otimes \beta_{i_2}$, and removes β_{i_1} and β_{i_2} from $\mathcal{C}$.
- At each step $j > 1$, let β_{j-1}^r be the intermediate result computed so far. The algorithm then selects and remove another curve from $\mathcal{C}$, let it be $\beta_{i_{j+1}}$, to compute $\beta_j^r = \beta_{j-1}^r \otimes \beta_{i_{j+1}}$.

After $n-1$ steps the set $\mathcal{C}$ is empty and β_{n-1}^r is the result of the (min,+) convolution. The permutation thus is determined by the order of choice: $\pi = (\beta_{i_1}, \beta_{i_2}, \beta_{i_3}, \ldots, \beta_{i_n})$.

The main benefit of such algorithm is that, while effects such as those of *representation minimization* [10] make it hard to predict the performance of a permutation π before running any computations, an iterative approach can adjust to the shape of the intermediate result β_i^r.

Key components of such algorithm are the way we estimate the runtime of each operation and the policies according to which we use the estimate to select the first pair $(\beta_{i_1}, \beta_{i_2})$ and next curve $\beta_{i_{j+1}}$, which we will refer to as *first pair* and *next curve* policies. These are discussed in the following Sections.

III. Runtime prediction

An important step of such an algorithm is predicting the computation time that an operation will take. We focused on estimating the runtime of the convolution of a given pair of curves, evaluating the accuracy vs. the overhead introduced by such estimation.

To do so, we need to give some details on how the (min,+) convolution is computed. As discussed in [12], [14], the (min,+) convolution algorithm is divided in two layers: a *by-curve* algorithm and a *by-sequence* algorithm. In the *by-curve* algorithm, the pseudo-periodic properties (pseudo-period length, pseudo-period height, etc.) of the operands are used to limit the amount of points and segments to be considered from each operand, as well as determine the pseudo-periodic properties of the result. In the *by-sequence* algorithm, the points and segments given from the previous layer are used to compute the points and segments of the result.

In practice, only a small fraction of the runtime is spent on the *by-curve* algorithm. The runtime of the *by-sequence* algorithm is related to the number of points and segments of the operands to be used: let these be N_f and N_g, the algorithmic complexity is $\mathcal{O}(N_f \cdot N_g \cdot \log(N_f \cdot N_g))$. The above expression is related to an execution of the algorithm that considers all possible pairs of points and segments from the two operands, $N_f \cdot N_g$, which is a metric that is easily computed. We call this metric *Element Product*.

However, due to various optimizations [12], [13], many of these pairs are unnecessary, and are not computed. So we can compute another metric, which we call *Operation Count*, which considers this to count only the pairs that the *by-sequence* algorithm would compute. However, computing this metric is by itself $\mathcal{O}(N_f \cdot N_g)$ which, for sequences with a high number of elements, is a non-negligible overhead.

To evaluate the effectiveness and prediction power of these metrics we used a test set of 5000 pairs of curves. Figure 2 shows the correlation between the measured computation time and the value predicted with *Operation Count*. As expected, the graph shows a clear correlation between the predicted and measured values, with a linear correlation coefficient of 0.986.

On the same test set, *Element Product* performs slightly worse, as shown in Figure 3. This metric struggles with extremely small sequences, probably because with such low numbers of operations even a single optimization can greatly change the result. Despite this, the correlation is still clear, with a linear correlation coefficient of 0.981, just 0.005 lower

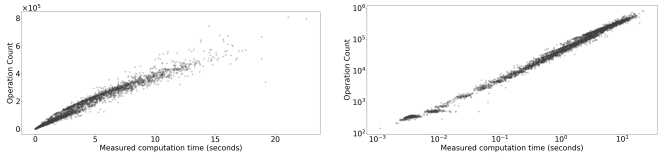

Fig. 2. Graphs showing measured time (x-axis) and prediction using *Operation Count* (y-axis) on both a linear (left) and a logarithmic (right) scale.

than *Operation Count*. The computation time required for the prediction is however much lower: on the testing dataset *Element Product* had on average a runtime of 0.47% of the actual computation, compared to 4.79% of *Operation Count*.

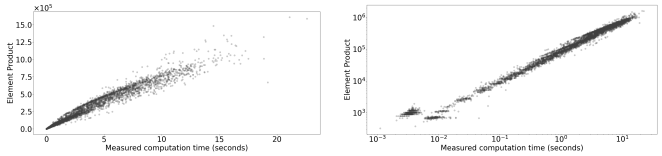

Fig. 3. Graphs showing measured time (x-axis) and prediction using *Element Product* (y-axis) on both a linear (left) and a logarithmic (right) scale.

IV. Heuristic reordering

In order to heuristically reorder the curves, taking advantage of the commutativity of (min,+) convolution, We need two policies: *first pair* and *next curve*. We envisioned these policies to be based on the runtime of single operations, with the overall goal of minimizing the time taken to compute all $n-1$ convolutions.

The strategy we followed was to list different policies that could be used for the two selections, and test all their possible pairings. Given a pair of policies and a set of curves, this gives us a permutation of curves and thus an order of operations, whose runtime lies somewhere in the x-axis of the distribution shown in Figure 1. We computed the percentile of the runtime of this permutation against all others: we say that a policy gives the 30th percentile on a set curves if the runtime of the (min,+) convolution of this set, using the permutation of this pair of policies, is higher than 30% of all other possible permutations (which implies faster than 70%). We thus compared the policies based on the percentile given on the sets of curves mentioned in Section II. Furthermore, the above evaluation was done using different runtime prediction metrics. To give a perspective on the accuracy of policies in ideal conditions and their degradation w.r.t. the accuracy of the runtime prediction, we first compare them using the *actual runtime*, which cannot be used in practice as it is unknown beforehand. Then, we will show how their performance vary using *Element Product* and *Operations Count*.

For *first pair* policies, we tested the following:

- *shortest first*: choose the pair that has the shortest runtime;
- *longest first*: choose the pair that has the longest runtime;
- *shortest between longest*: for each curve, choose the pair that contains it with the longest runtime, then among these pairs choose the one with the shortest runtime;
- *median of medians*: for each curve, choose the pair that contains it with the median runtime, then among these pairs choose the one with the median runtime;
- *shortest median*: for each curve, choose the pair that contains it with the median runtime, then among these pairs choose the one with the shortest runtime.

For *next pair* policies, we tested the following:

- *shortest-first*: choose the curve that, paired β^r_{j-1}, has the shortest runtime;
- *longest-first*: choose the curve that, paired β^r_{j-1}, has the longest runtime;
- *median*: choose the curve that, paired β^r_{j-1}, has the median runtime;

The first observation that we gathered is that for *next curve* there is a clear winner in the *shortest first* policy, since it performs better than the other two options in all pairings with *first pair* policies, as well as all sets of curves. In Table I we report the comparison between *next curve* policies when paired with *shortest between longest*, similar results are found with other *first pair* policies.

Table I
Percentile given by different *next curve* policies, using *actual runtime* and *shortest between longest first pair* policy.

Next Pair Algorithm	Average	Max	Min	Standard Deviation
Shortest-First	28	68	2	24
Longest-First	73	99	6	33
Median	58	95	14	29

Instead, in *first pair* policies the comparison was more interesting. We observed that selecting pairs that produce either too slow or too fast convolutions tends to give worse results than choosing something in the middle. This is intuitive for the *longest first* policy: in many cases, the longest runtime for the first convolution is, by itself, larger than the average for the entire chain over all permutations. It is instead counter-intuitive for the *shortest first* policy: a possible explanation is that selecting the simplest curves first forces a worse pairing later in the chain. We found instead better performance in the following policies, designed to select something "in the middle", i.e., *shortest between longest*, *median of medians* and *shortest median*. Since, as discussed above, the *next curve* policy *shortest first* performs better than all others heuristics we tried, the following comparisons will all be made using it.

Table II shows the comparison of the different policies by percentiles. To give a perspective on the time saved (or not), Table III reports the same as ratio over the average runtime over all permutations. The sets of curves that we used highlight different behaviors: in some outlier instances the best ordering is easily found by our policies, while in others they point to suboptimal permutations. On average, we observed a 30% reduction of runtime from the use of these policies. More importantly, we observed that all policies are effective in avoiding the right tail of the distribution, which, as Figure 1 shows, can reach over 2.9 times the average.

Table II
Percentile given by different *first pair* policies, using *actual runtime*.

First Pair Algorithm	Average	Max	Min	Standard Deviation
Shortest	40	72	7	22
Longest	44	81	11	27
Shortest between longest	28	68	2	24
Median of medians	22	73	2	24
Shortest median	32	73	2	23

Table III
Ratio of runtime given by different *first pair* policies (using *actual runtime*) over average across all permutations.

First Pair Algorithm	Average	Max	Min	Standard Deviation
Shortest	0.783	1.136	0.486	0.219
Longest	0.916	1.401	0.588	0.318
Shortest between longest	0.643	1.174	0.324	0.294
Median of medians	0.656	1.289	0.282	0.312
Shortest median	0.682	1.144	0.324	0.254

Tables IV and V show instead how these policies perform w.r.t. the accuracy of the runtime estimation metric being used, where *actual runtime* acts as an ideal case with maximum estimation accuracy. From these comparisons, we see that *shortest between longest* behaved better than the others, as it performs well when using estimates, while *median of medians* performs well only in the ideal case.

On the topic of runtime estimates, we can see from these results that the difference in accuracy between *Operation Count* and *Element Product* are not reflected in the policy performance. Note that Tables IV and V do not take into account the overhead induced by estimates, which we measured to be, compared to runtime for the whole chain convolution, between 10% and 19% for *Operation Count* and between 0.1% and 0.3% for *Element Product*. This strongly suggest that using *Element Product* is sufficient for our purposes.

Table IV
Average percentile given by different *first pair* policies, using different runtime estimation methods.

First Pair Algorithm	Actual Runtime	Operation Count	Element Product
Shortest	40	40	40
Longest	44	46	51
Shortest between longest	28	28	28
Median of medians	22	29	29
Shortest median	32	31	31

We can visualize the impact of these results using the same example of Figure 1. In Figure 4, we can see that using our heuristics one can obtain better performance by not leaving the ordering to chance.

V. Conclusion

In this paper, we have shown that, while the (min,+) convolution is commutative and associative, its runtime does change based on the order in which operations are performed. We thus explored an algorithm that uses heuristics to reorder the curves based on runtime prediction, showing that it is able to avoid the cases with higher runtimes, and reduce average runtime by 30%.

Table V
Average ratio of runtime given by different *first pair* policies over average across all permutations, using different runtime estimation methods.

First Pair Algorithm	Actual Runtime	Operation Count	Element Product
Shortest	0.783	0.783	0.783
Longest	0.916	1.009	1.043
Shortest between longest	0.643	0.643	0.643
Median of medians	0.656	0.704	0.704
Shortest median	0.682	0.672	0.672

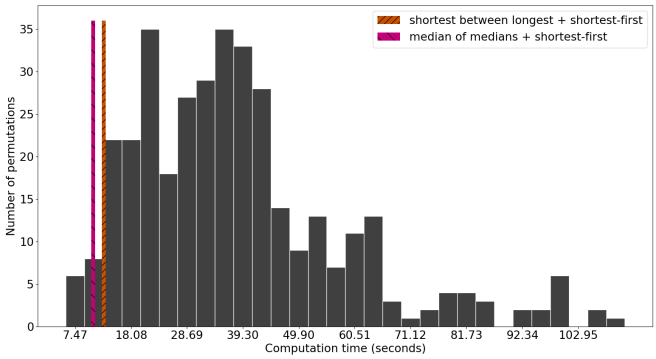

Fig. 4. Distribution of runtimes to compute the (min,+) convolution of a set of 6 curves, according to all possible permutations of the set, vs. the runtimes obtained using our heuristics.

While promising, there are many paths for improvement for this approach. One such direction is further research on the algebraic properties that affect runtime, in particular those that affect the efficacy of optimizations from [10], [12]–[14]. For example, our approach would benefit from a way to predict the runtime on a permutation basis rather than only on a pair to pair basis.

Another direction is the extension of the heuristic approach to exploit the associativity of (min,+) convolution, as well. In this work we focused on the commutativity of the (min,+) convolution due to the explosive increase in number of computations needed to construct a dataset to evaluate heuristics. In fact, to compute a chained convolution of n curves using commutativity we have $\mathcal{O}(n!)$ many ways to reorder the operands and obtain different runtimes; but if associativity is also considered then we have $\mathcal{O}(\frac{(2n-2)!}{(n-1)!})$ to do so. Even for $n = 6$, this is tens of thousands of computations compared to the few hundreds we discussed in this paper. Given these initial results, it is however a promising direction to follow in future work.

References

[1] R. L. Cruz, "A calculus for network delay, part I: Network elements in isolation," *IEEE Transactions on information theory*, vol. 37, no. 1, pp. 114–131, 1991.

[2] R. L. Cruz, "A calculus for network delay, part II: Network analysis," *IEEE Transactions on information theory*, vol. 37, no. 1, pp. 132–141, 1991.

[3] C.-S. Chang, *Performance guarantees in communication networks*. New York, USA: Springer-Verlang, 2000.

[4] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Berlin, Germany: Springer Science & Business Media, 2001.

[5] A. Bouillard, M. Boyer and E. Le Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*. Hoboken, NJ: Wiley, 2018.

[6] L. Thiele, S. Chakraborty and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2000, 101–104 vol.4. DOI: 10.1109/ISCAS.2000.858698.

[7] F. Baccelli, G. Cohen, G. J. Olsder and J.-P. Quadrat, "Synchronization and linearity: An algebra for discrete event systems," 1992.

[8] 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, ISSN: 15540588. DOI: 10.1561/1300000059.

[9] Y. Tang, Y. Jiang, X. Jiang and N. Guan, "Pay-burst-only-once in real-time calculus," in *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019, pp. 1–6. DOI: 10.1109/RTCSA.2019.8864582.

[10] R. Zippo and G. Stea, "Computationally efficient worst-case analysis of flow-controlled networks with Network Calculus," *IEEE Transactions on Information Theory*, 2023. DOI: 10.1109/TIT.2023.3244276.

[11] R. Zippo, P. Nikolaus and G. Stea, "Extending the Network Calculus Algorithmic Toolbox for Ultimately Pseudo-Periodic Functions: Pseudo-Inverse and Composition," *Discrete Event Dynamic Systems*, 2023, ISSN: 1573-7594. DOI: 10.1007/s10626-022-00373-5. [Online]. Available: https://link.springer.com/article/10.1007/s10626-022-00373-5.

[12] R. Zippo, P. Nikolaus and G. Stea, "Isospeed: Improving (min,+) convolution by exploiting (min,+)/(max,+) isomorphism," in *35th Euromicro Conference on Real-Time Systems (ECRTS'23)*, IEEE, 2023, 24–pp. DOI: 10.4230/LIPIcs.ECRTS.2023.

[13] R. Zippo, "Analysis of algorithmic and computational aspects of deterministic network calculus," 2023.

[14] A. Bouillard and É. Thierry, "An algorithmic toolbox for network calculus," *Discrete Event Dynamic Systems*, vol. 18, no. 1, pp. 3–49, 2008.

[15] E. Wandeler and L. Thiele, *Real-Time Calculus (RTC) Toolbox*, `http://www.mpa.ethz.ch/Rtctoolbox`. [Online]. Available: http://www.mpa.ethz.ch/Rtctoolbox (visited on 01/11/2021).

[16] RealTime-at-Work, *RTaW-Pegase (min,+) library*, https://www.realtimeatwork.com/rtaw-pegase-libraries/, Accessed: 2022-04-05.

[17] R. Zippo and G. Stea, "Nancy: An efficient parallel Network Calculus library," *SoftwareX*, vol. 19, p. 101 178, 2022, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2022.101178. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235271102200108X.

[18] N. Guan and W. Yi, "Finitary real-time calculus: Efficient performance analysis of distributed embedded systems," in *2013 IEEE 34th Real-Time Systems Symposium*, 2013, pp. 330–339.

[19] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan and W. Yi, "Generalized finitary Real-Time calculus," in *Proc. of the 36th IEEE International Conference on Computer Communications (INFOCOM 2017)*, 2017.

[20] S. M. Tabatabaee, M. Boyer, J.-Y. B. Le and J. Migge, "Efficient and accurate handling of periodic flows in time-sensitive networks," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 303–315. DOI: 10.1109/RTAS58335.2023.00031.