A Configurable Hardware Architecture for Runtime Application of Network Calculus

Xiao Hu, Zhonghai Lu

▶ **To cite this version:**

Xiao Hu, Zhonghai Lu. A Configurable Hardware Architecture for Runtime Application of Network Calculus. 17th IFIP International Conference on Network and Parallel Computing (NPC), Sep 2020, Zhengzhou, China. pp.203-216, 10.1007/978-3-030-79478-1_18 . hal-03768725

HAL Id: hal-03768725
https://inria.hal.science/hal-03768725

Submitted on 4 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

A Configurable Hardware Architecture for Runtime Application of Network Calculus

Xiao Hu [1] and Zhonghai Lu [2]

[1] School of Computer, National University of Defense Technology, Changsha, P. R. of China
[2] KTH Royal Institute of Technology, Stockholm, Sweden
[1] xiaohu@nudt.edu.cn, [2] zhonghai@kth.se

Abstract. Network Calculus has been a foundational theory for analyzing and ensuring Quality-of-Service (QoS) in a variety of networks including Networks on Chip (NoCs). To fulfill dynamic QoS requirements of applications, runtime application of network calculus is essential. However, the primitive operations in network calculus such as arrival curve, min-plus convolution and min-plus deconvolution are very time consuming when calculated in software because of the large volume and long latency of computation. For the first time, we propose a configurable hardware architecture to enable runtime application of network calculus. It employs a unified pipeline that can be dynamically configured to efficiently calculate the arrival curve, min-plus convolution, and min-plus deconvolution at runtime. We have implemented and synthesized this hardware architecture on a Xilinx FPGA platform to quantify its performance and resource consumption. Furthermore, we have built a prototype NoC system incorporating this hardware for dynamic flow regulation to effectively achieve QoS at runtime.

Keywords: Network Calculus, Hardware Architecture, Hardware Configuration, Network-on-Chip, Quality-of-Service.

1 Introduction

Network Calculus [1][2][3][4] has been an active research area and successfully applied to fulfill Quality-of-Service (QoS) requirements of various networks. Recently, it has also been successfully applied to Networks on Chip (NoCs) in Chip Many-core Processors (CMPs) and Many-Processor Systems-on-Chip (MPSoCs) [5][6][7][8].

Traditionally, network calculus is used at design time as a theoretical tool for worst-case performance derivations of packet delay upper bound, maximum buffer backlog, minimal flow throughput etc. In recent years, network calculus is also applied in dynamic network admission control to monitor the changing traffic scenario in hard real-time systems. Huang *et al.* proposed a light-weight hardware module to address the traffic conformity problem for run-time inputs of a hard realtime system [6]. The arrival curve capturing the worst-case/best-case event arrivals in the time domain can be conservatively approximated by a set of staircase functions, each of which can be modeled by a leaky bucket. They used a dual-bucket mechanism to monitor each staircase function during run-time, one for conformity verification and the other for traffic regulation. In case too many violation events are detected, the regulator delays

the input events to fulfill the arrival curve specification assumed at design time. By conducting the conformity check, the system is able to monitor and regulate the actual behavior of NoC traffic flows in order to realize dynamic QoS in time-critical applications. However, the method in [6] for the conformity check of actual traffic stream against predefined specification assumes the linear arrival curve. Since it does not compute the arrival curve, it cannot be used for general arrival curve. Also, to enable a full scale of applying network calculus for dynamic QoS assurance, a systematic approach needs to be taken. For example, to compute output arrival curve needs to realize min-plus deconvolution, because output arrival curve is the result of min-plus deconvolution between input arrival curve and service curve. Indeed, to process both arrival curve and service curve, we need to calculate basic network calculus operations, which include both min-plus convolution and min-plus deconvolution.

From the software perspective, basic network calculus operations such as arrival curve, min-plus convolution, and min-plus deconvolution can be computed at runtime but are very time-consuming due to high complexity. For example, the computation complexity of the min-plus deconvolution operation in the recursive Equation (8) (Section 3.3) is $O(N)$, where N is the length of calculation window in number of data items or cycles. When N=128, there are 256 (128×2) operations for computation. In software, it costs about 22.9 microseconds on an Intel Core i3-3240 3.4GHz CPU with Windows 7 operating system (see Section 4.3). However, in timing-critical applications, the system requires quick verification and fast regulation of flows online in several cycles according to the computation results of network calculus. Under such circumstances, how to accelerate the calculation speed in hardware fully supporting network calculus operations becomes an open challenge. Furthermore, to be efficient, it is desirable to have the network calculus hardware architecture configurable such that different operations can be done by simple configurations on the same hardware substrate.

To address the above challenge, we propose a hardware architecture for runtime (online) computation of network calculus operations. This hardware architecture is designed by analyzing the rudimentary definitions of the arrival curve, min-plus convolution and min-plus deconvolution. Through analyzing their recursive accumulative behaviors in their mathematical representations, we are able to reckon a unified pipeline architecture to conduct these primitive operations through simple configurations via de-multiplexing and multiplexing selections. We have implemented and optimized the hardware design and synthesized it on FPGA. In a case study, the specialized hardware module is used to build a runtime flow monitor attached to regulators in the network interface of NoC so as to facilitate dynamic flow regulation. To the best of our knowledge, no previous research has touched upon this approach.

The main contributions of the paper can be summarized as follows.

1. We develop a configurable hardware architecture for runtime computation of network calculus operations including arrival curve, min-plus convolution, and min-plus deconvolution. The hardware architecture features a unified pipeline where the three network calculus operations can be performed by runtime configurations.

2. We implement the proposed design on a Xilinx FPGA platform and evaluate its area and speed, demonstrating its efficiency and feasibility.
3. With a multi-media playback system, the architecture is prototyped and used to satisfy application QoS, showing its potential in runtime monitoring of QoS bounds.

2 Related Work

Network calculus originated from macro networks for performance guarantees in Internet and ATM [1][2][3][4]. Theoretically it transforms complex non-linear network systems into analyzable linear systems [2][3][4]. In real-time calculus [9], it is extended to define both upper/lower arrival curves and upper/lower service curves to compute worst-case delay bounds under various scheduling polices for real-time tasks.

In recent years, network calculus has been applied to NoCs for analyzing worst-case performance guarantees of real-time applications, for example, to determine the worst-case reorder buffer size [11], to design network congestion control strategy [13] and to develop a per-flow delay bound analysis methodology for Intel's eXtensible Micro-Architectural Specification (xMAS) [7]. Notably in industrial practices, network calculus has been employed as a theoretical foundation to build the data NoC of Kalray's MPPA-256 many-core processor to achieve guaranteed communication services in per-flow delay and bandwidth [8].

In network calculus, traffic specification (e.g. linear arrival curve) can be used not only to characterize flows but also to serve as a contract for QoS specification. Subsequently, flow regulation as a traffic shaping technique can be employed at runtime for admission control to check conformity. In [10][12], flow regulation is used to achieve QoS communication with low buffering cost when integrating IPs to NoC architectures. Lu and Wang presented a dynamic flow regulation [12], which overcomes the rigidity of static flow regulation that pre-configures regulation parameters statically and only once. The dynamic regulation is made possible by employing a sliding window based runtime flow (σ, ρ) characterization technique, where σ bounds traffic burstiness and ρ reflects the average rate. The effectiveness of dynamic traffic regulation for system performance improvement is further demonstrated in [14].

3 Configurable Hardware Architecture

We consider network calculus in a digital system. A data packet stream, noted as *flow*, arrives cycle by cycle. The two basic operations in network calculus are *min-plus convolution* and *min-plus deconvolution* [2] in min-plus algebra, noted as $f \otimes g$ and $f \oslash g$, respectively (see definitions below). There are two input functions, f and g, in convolution and deconvolution. When the two functions are the same, they are noted as $f \otimes f$ and $f \oslash f$, respectively. The result of $f \oslash f$ is in fact the *Arrival Curve (**AC**)* [2], which may be separated as the third operation due to its importance.

To conduct network calculus calculations in hardware at system runtime, we propose a unified hardware architecture that can flexibly support all above three basic network

calculus operations, i.e., $f \otimes g$, $f \oslash g$, and $f \oslash f$ (***AC***) by simple configurations. As such, the hardware resources consumed by these operations can be shared for efficiency so as to facilitate and justify runtime application of network calculus. In the following, we detail our flexible hardware architecture step by step.

3.1 Micro-architecture for function $f \oslash f$ (Arrival Curve)

We start by designing a functional hardware architecture for Arrival Curve.

Definition of arrival curve [2]: Given a wide-sense increasing function α defined for $t \geq 0$, we say that a flow f is constrained by α if and only if for all $s \leq t$: $f(t) - f(s) \leq \alpha(t - s)$. Equivalently we say that f has α as an arrival curve.

Let d_i be the size of arrival data at cycle i, from the definition, we have:

$$(f \Phi f)(t) = \sup_{u \geq 0}\{f(t+u) - f(u)\} = \sup_{u \geq 0}\{\textstyle\sum_{u+1 \leq i \leq t+u} d_i\},\ t > 0 \quad (1)$$

Here ***sup*** is the supremum operator. We can define $\sum_{u+1 \leq i \leq t+u} d_i$ in Equation (1) as an intermediate function, named *AR(t)*. Then we have

$$(f \Phi f)(t) = \sup_{u \geq 0}\{AR(t)\} \quad (2)$$

Furthermore, *AR(t)* can be iteratively calculated by the following recursive function:

$$AR(t) = \sum_{u+1 \leq i \leq t+u} d_i = \sum_{u+1 \leq i \leq t+u-1} d_i + d_{t+u} = d_{t+u} + AR(t-1) \quad (3)$$

In particular, $AR(0) = d_0$

Compute micro-architecture for arrival curve: We take advantage of the recursive equation of *AR(t)* in Equation (3) to define an effective hardware micro-architecture for computing arrival curve. We can observe that, by defining cascaded registers storing *AR(t)* values, $(f \Phi f)(t)$ can be transformed into recording the maximum values in *AR(t)* registers. In this way, we can design a pipeline circuit to efficiently calculate the arrival curve in a processing window to handle the continuous data stream.

Fig. 1 draws the hardware micro-architecture for computing arrival curve. The basic logic unit is called AddShiftComp unit. There are *N* AddShiftComp units cascaded in a pipeline. Each unit has an adder, a comparator, a multiplexer and a shifter connected to the next unit. As a generic efficient hardware design, the arrival curve is only calculated in one sliding window with a length of *N* data items. The Sampling unit is used in the sampling mode, which is to be detailed in the next section. If the Sampling unit is bypassed, a new data item flows into the processing pipeline at each cycle.

In Fig. 1, *f(t)* is the input flow and d_i is the volume of arrival data at cycle *i*. AR is the Accumulating Register and BR is the Bound Register. The circuit also comprises the adders, comparators and multiplexers. On each cycle, the value of every AR added with current d_i is written into the next AR. Each AR is compared with the corresponding BR, the bigger one is written into the BR again. The Samp_CTL, AR_RST and BR_RST are control signals. The Samp_CTL is designed for the sampling mode. The AR_RST and BR_RST are used to initialize the AR and BR registers. After *N*+2 cycles, the values of the accumulating function in the window of length N are computed and stored in ARs. The maximum bound values of ARs are stored in BRs. All values in BRs represent the arrival curve. The results in BRs can be snapshotted or registered and shifted out. When

clearing all ARs and BRs with related Reset (RST) signals, the process restarts. To use the results in BRs, i.e., the dynamic arrival curve, by other circuits, we design Snapshot&Shiftout registers (SFs). With Control signal, these SFs are updated with all BRs snapshoted and shifted out one by one.

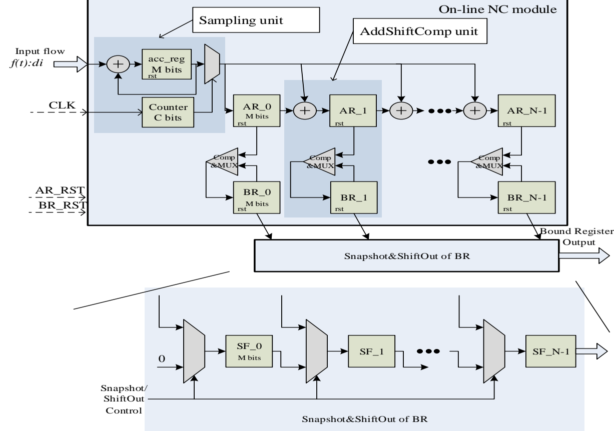


Fig. 1. Hardware micro-architecture for computing arrival curve.

Operation details with an example: The process of computing arrival curve is listed in Fig. 2. Taking N = 4 as an example, the processing details are given in Table 1. As N=4, there are 4 ARs (AR_0~AR_3) and 4 BRs (BR_0~BR_3). At cycle 1, the volume of arrival data is d_0 and all ARs and BRs are cleared with AR_RSTand BR_RST. At cycle 2, the volume of arrival data is d_1 and all ARs are d_0 and all BRs are still 0. As the cycle time advances, AR_0 is equal to last data item d_{i-1}, AR_1 equal to $d_{i-2}+d_{i-1}$, AR_2 equal to $d_{i-3}+d_{i-2}+d_{i-1}$ and AR_3 equal to $d_{i-4}+d_{i-3}+d_{i-2}+d_{i-1}$. BR_0 stores the maximum value of AR_0, i.e., $\sup_{0\le j\le i-2}\{d_j\}$. BR_1 stores the maximum value of AR_1, i.e., $\sup_{0\le j\le i-3}\{d_j+d_{j+1}\}$. BR_2 stores the maximum value of AR_2, which is $\sup_{0\le j\le i-4}\{d_j+d_{j+1}+d_{j+2}\}$. BR_3 stores the maximum value of AR_3, which is $\sup_{0\le j\le i-5}\{d_j+d_{j+1}+d_{j+2}+d_{j+3}\}$. Then we get arrival curve via BR_0 ~ BR_3.

The hardware cost can be estimated from Fig. 1 (2×N×M register bits for AR/BR and 2×N adders for compare/add). It is almost linear with number N of AddShiftComp units.

1: //Config step
2: Clear all ARs with AR_RST
3: Clear all BRs with BR_RST
4: //Work step
5: while (N)
6: { input d_i per cycle }
7: //Output step
8: Snapshot BR Registers into Snapshot&Shiftout registers
9: Shift out Snapshot&Shiftout registers (Arrival Curve) to other circuits one by one

Fig. 2. Process of computing arrival curve.

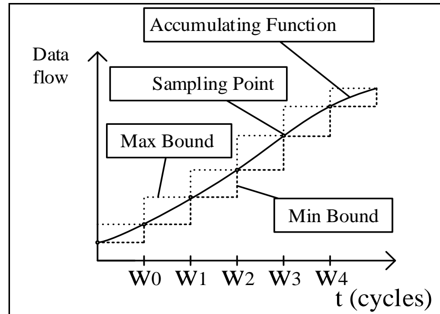


Fig. 3. Sampling-mode bounds for Arrival Curve.

Table 1. Register details of computing arrival curve

Cycle	In	AR_0	AR_1	AR_2	AR_3	BR_0	BR_1	BR_2	BR_3
	d_0	0	0	0	0	0	0	0	0
	d_1	d_0	$0 + d_0$	$0 + 0 + d_0$	$0 + 0 + 0 + d_0$	0	0	0	0
	d_2	d_1	$d_0 + d_1$	$0 + d_0 + d_1$	$0 + 0 + d_0 + d_1$	$\sup\{ d_0\}$	d_0	d_0	d_0
	d_3	d_2	$d_1 + d_2$	$d_0 + d_1 + d_2$	$0 + d_0 + d_1 + d_2$	$\sup\{ d_0, d_1\}$	$\sup\{ d_0 + d_1\}$	$d_0 + d_1$	$d_0 + d_1$
	d_4	d_3	$d_2 + d_3$	$d_1 + d_2 + d_3$	$d_0 + d_1 + d_2 + d_3$	$\sup\{ d_0, d_1, d_2\}$	$\sup\{ d_0 + d_1, d_1 + d_2\}$	$\sup\{ d_0 + d_1 + d_2\}$	$d_0 + d_1 + d_2$
	d_5	d_4	$d_3 + d_4$	$d_2 + d_3 + d_4$	$d_1 + d_2 + d_3 + d_4$	$\sup\{ d_0, d_1, d_2, d_3\}$	$\sup\{ d_0 + d_1, d_1 + d_2, d_2 + d_3\}$	$\sup\{d_0 + d_1 + d_2, d_1 + d_2 + d_3\}$	$\sup\{d_0 + d_1 + d_2 + d_3\}$
...	...	...	...	...	...	...	...	...	...

3.2 Sampling-based micro-architecture for arrival curve

For some applications, there is a need to sample arrival curve at a larger time granularity than per cycle. For example, a system might not generate input data at each and every cycle. It is possible that the traffic generation is asynchronous and has a larger period than the arrival curve computation hardware. It might also be possible that an arrival curve at a larger time granularity is more interesting for the QoS analysis. In such cases, a larger time scale is needed to calculate arrival curve. To support this feature, we design a sampling scheme at a larger time scale as the sampling module shown in Fig. 1. It consists of a *C-bit* counter, an acc_reg register and an accumulator. Input d_i is accumulated into the acc_reg every cycle continuously. The *C-bit* counter as a controller enables the acc_reg output to the pipeline at a period of W cycles. The circuit samples the arrival curve every W cycles in the sampling mode (the ith sampling point is at $i \times W$ cycles). The max/min bound is indicated by the upper/lower stairs in Fig. 3.

Comparing with the original scheme recording all data of Full Accumulating Function (FAF) curve, the accumulating function curve recorded in the sampling mode (Sampling Accumulating Function, SAF) is composed of these sampling points. The SAF is accurate at these sampling points. Between two sampling points, the FAF may be any curve not larger than the upper sampling point and not less than the lower sampling point. Therefore, the maximum bound of FAF is the upper stairs set by sampling points and the minimum bound of FAF is the lower stairs set by sampling points.

The maximum bound of arrival curve can be expressed as:

$$\alpha_{\max} = \begin{cases} \sup\limits_{0 \le i}\{w_i + w_{i+1}\}, 0 \le t < T \\ \sup\limits_{0 \le i}\{w_i + w_{i+1} + w_{i+2}\}, T \le t < 2T \\ \sup\limits_{0 \le i}\{w_i + w_{i+1} + w_{i+2} + w_{i+3}\}, 2T \le t < 3T \\ \dots \end{cases} \tag{4}$$

The minimum bound of arrival curve can be expressed as:

$$\alpha_{\min} = \begin{cases} \sup\limits_{0 \le i}\{ w_i \}, 0 \le t < T \\ \sup\limits_{0 \le i}\{w_i + w_{i+1}\}, T \le t < 2T \\ \sup\limits_{0 \le i}\{w_i + w_{i+1} + w_{i+2}\}, 2T \le t < 3T \\ \dots \end{cases} \tag{5}$$

3.3 Micro-architecture for function *f∅g*

Definition of min-plus deconvolution [2]: *f∅g* denotes the min-plus deconvolution. Let *f* and *g* be two functions or sequences. The min-plus deconvolution of *f* by *g* is the function: $(f\Phi g)(t) = \sup_{u\geq 0}\{f(t+u) - g(u)\}$ (6)

Compared to common convolution, min-plus deconvolution uses the *maximum* respectively *supremum* (*sup*) operator to replace the *sum* operator and the *minus* operator to replace the *product* operator. Assume that *f(t)* and *g(t)* are two infinite data flows denoted by d_i and e_i, respectively. Time t is in clock cycle. From the definition of function *f∅g*, we have: $(f\Phi g)(t) = \sup_{u\geq 0}\{\sum_{0\leq i\leq t+u} d_i - \sum_{0\leq i\leq u} e_i\}$ (7)

We can define *AR(t)* in the same way as in Section 3.1:

$$AR(t) = \sum_{0\leq i\leq u+t} d_i - \sum_{0\leq i\leq u} e_i = d_{u+t} + \sum_{0\leq i\leq u+t-1} d_i - \sum_{0\leq i\leq u} e_i = d_{t+u} + AR(t-1) \quad (8)$$

For *AR(0),* we have: $AR(0) = \sum_{0\leq i\leq u} d_i - \sum_{0\leq i\leq u} e_i = \sum_{0\leq i\leq u}(d_i - e_i)$ (9)

Compute micro-architecture for *f∅g*: Since Equation (8) is similar to Equation (3), this means that we can reuse and enhance the hardware micro-structure for *f∅f* to realize the general *f∅g* operation. Specifically, an *SubAcc unit* is added to the input part of the hardware circuit of *f∅f* to calculate function *f∅g*, as shown in Fig. 4. When *f(t)=g(t)*, the diff_reg and AR_0 register are always zero in the *SubAcc unit* so they can be omitted and the circuit turns into *f∅f* with *N-1* items (BR_0 is always zero).

With the function in Equation (7), for a *f∅g* curve with $t = N$ cycles, the total calculation operations are $N\times(1+3+5+\ldots+(2u+1))$. When $u = N$, the computation complexity of the deconvolution operation is $O(N^2)$.

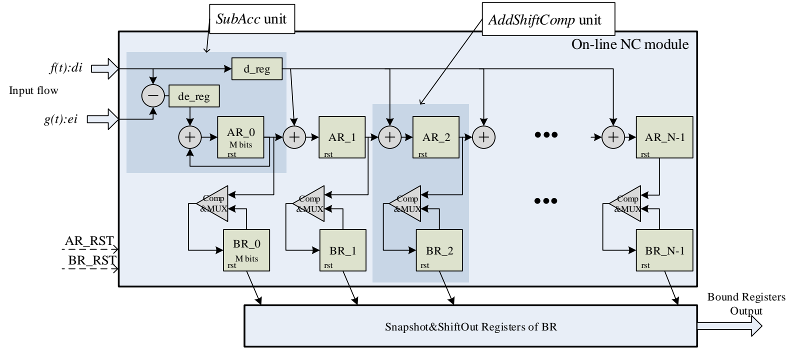


Fig. 4. Hardware micro-architecture for computing *f∅g*

3.4 Micro-architecture for function *f⊗g*

Definition of min-plus convolution [2]: Let *f* and *g* be two functions or sequences. The min-plus convolution of *f* and *g* denoted by *f⊗g* is the function

$$(f \otimes g)(t) = \inf_{0\leq s\leq t}\{f(t-s) + g(s)\} \quad (10)$$

Compared to common convolution, min-plus convolution uses the *minimum* respectively *infimum* (***inf***) operator to replace the *sum* operator and the *sum* operator to

replace the *product* operator. Suppose that *g(t)* is an infinite data flow denoted by e_i and *f(t)* denoted by d_i.

$$(f \otimes g)(t) = \inf_{0 \le s \le t} \{\sum_{0 \le i \le t-s} d_i + \sum_{0 \le i \le s} e_i\} \quad (11)$$

Again, we can define *AR(t)* in the same way as in Section 3.1:

$$AR(t) = \sum_{0 \le i \le t-s} d_i + \sum_{0 \le i \le s} e_i = d_{t-s} + \sum_{0 \le i \le t-s-1} d_i + \sum_{0 \le i \le s} e_i = d_{t-s} + AR(t-1) \quad (12)$$

For *AR(0)*, we have: $AR(0) = d_0 + e_0$ (13)

Compute micro-architecture of $f \otimes g$: Since Equation (12) is similar to Equation (3), we can reuse and enhance the hardware micro-structure for $f \oslash f$ to realize the $f \otimes g$ operation. Specifically, a Mux unit is added to the hardware circuit of $f \oslash f$ to deal with two inputs of *g(t)* and *f(t)*, as shown in Fig. 5. There are two stages (Initial and Normal) when calculating function $f \otimes g$. The Initial Stage is to initialize the AR registers with *g(t)* (input flow is e_{N-1}, e_{N-2},,,e_1,e_0 cycle by cycle) by setting the control signal Mux_CTL. After the Initial Stage, the content of the ith AR register is *g(i)* ($\sum_{0 \le j \le i} e_j$). The Normal Stage is to compute the function $f \otimes g$ by setting the control signal Mux_CTL to the *f(t)* channel. The comparator is configured such that the smaller one of the two inputs is written into the BR register for the ***inf*** operation. The BL_CTL signals are added to enable each of the comparators to remove useless comparison results. The register content details for computing function $f \otimes g$ are similar to Table 1.

3.5 Unified micro-architecture with function configuration

Combining these hardware micro-architectures by switches, we obtain a *unified configurable* hardware architecture for executing the network calculus functions as drawn in Fig. 5. The shared part is the central pipeline with AddShiftComp units, each of which contains 2 *M-bit adders* and 2 *M-bit registers*. Different network calculus operations are realized by adding switches on the Sampling unit (from the arrival curve unit in Fig. 1), SubAcc unit (from the $f \oslash g$ unit in Fig. 4) and Mux unit (from the $f \otimes g$ unit). When configured to the arrival curve mode, d_i is switched to AddShiftComp units through the Sampling unit directly. When configured to the $f \oslash g$ mode, d_i and e_i are switched to the SubAcc unit through each sampling unit. When configured to the $f \otimes g$ mode, d_i and e_i are switched to the Mux unit through each sampling unit.

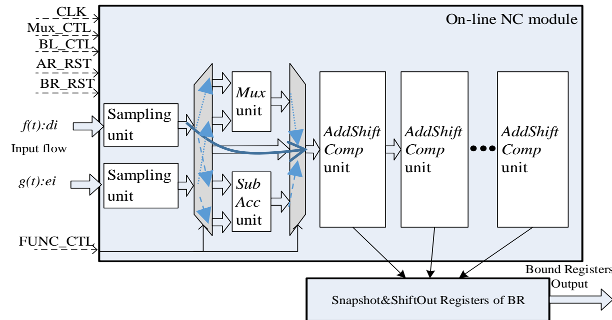


Fig. 5. Unified configurable pipeline hardware architecture for network calculus operations. The solid line is the datapath of configuration for arrival curve. (Dotted line: $f \oslash g$. Dashed line: $f \otimes g$)

The configurable hardware architecture generates results in one cycle because the N AddShiftComp units process data in parallel. In terms of resources, it costs only 1/3 of the non-configurable architecture which otherwise uses three individual hardware micro-architectures for the three network calculus functions. For a configurable hardware architecture with N units of AddShiftComp, the circuit only requires $2 \times N$ adders and $2 \times N$ registers with M-bit width. Thus, the hardware complexity is $O(N)$.

4 FPGA IMPLEMENTATION AND EVALUATION

We implemented the unified configurable pipeline hardware architecture on ZYNQ FPGA from Xilinx. The number of AddShiftComp units is N, the width of AR/BR register is M bits and the counter of sampling unit is C bits.

We validated the three basic network calculus operations with models realized in MATLAB. When using the same sampling method and no overflows, the results of FPGA and MATLAB implementations are the same, because the configurable hardware architecture is designed accurately according to the recursive equations.

4.1 Performance optimization

We further optimized the performance of the hardware design. Since the critical path of the circuit is the comparing and multiplexing of AR and BR, an additional register is inserted to the output of each comparator to shorten the critical path. Since the data path of input d_i to each adder has a big fan-out, an output register is added to the multiplexor.

Table 2 lists the FPGA implementation results ($N = 128$, $M = 16$) before and after the optimization. As can be seen, the register utilization is increased after the optimization. The total resources of LUT decrease by 25.2%. The frequency increases by 10.1%.

Table 2. Hardware implementation results. (AddShiftComp (N)=128, AR/BR register (M)=16)

Before Optimization		**After Optimization**	
Type	**Value**	**Type**	**Value**
LUT	6541	LUT	5222
registers	4096	registers	4112
Max Frequency	244.4 MHz	Max. Frequency	269.1 MHz

4.2 Scalability and overhead

The required resource utilizations and the maximum frequencies of different design parameters (N AddShiftComp units and M bits width) are evaluated. As shown in Fig. 6, the required resource utilizations increase linearly and the maximum frequencies are stable around 250MHz ~ 280MHz in the ZYNQ FPGA platform. These results show good scalability of the hardware architecture. When N=64&128 , the maximum frequency of M=16 is a bit larger than M=24 and M=32 . This is because the FPGA resources for logic synthesis of M=16 can be limited in one hardware block region.

When using 128 AddShiftComp units and 16-bit width AR/BR registers, the FPGA resource of the configurable hardware architecture is about 6k LUTs. Compared with the area-overhead of a recent flow generator & monitor in [15], our configurable hardware architecture is acceptable. When computing the arrival curve with N=128, it

takes 3.7 ns on 269.1 MHz frequency to generate the result. With parallel computing in hardware, the execution time of the proposed circuit only depends on the maximum frequency. This means no matter how big N gets, it costs about the same time.

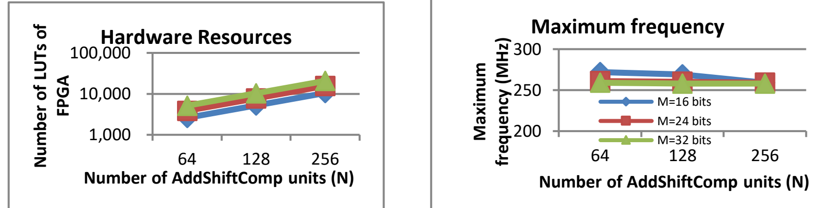


Fig. 6. Hardware resources and maximum frequencies on different design parameters

4.3 Comparison with software implementation

The Network Calculus such as arrival curve is computed only by software traditionlly. To obtain the speedup achieved by the specific hardware design, we realize an algorithm written in C language to do the arrival curve computation in software following the recursive function in Equations (2)(3). The computer has an Intel Core i3-3240 CPU running at 3.4 GHz frequency. The operating system is Windows 7. With the same parameter as for the FPGA hardware, the length N for the arrival curve computation is set to 128. Completing the 128×2 calculation operations (comparison and addition) in Equations (2)(3) takes 22.9 microseconds (memory accesses of CPU and the OS take most of the time). In contrast to the 3.7ns execution time in hardware, the hardware speedup is more than 6000 times.

5 SYSTEM PROTOTYPE AND CASE STUDY

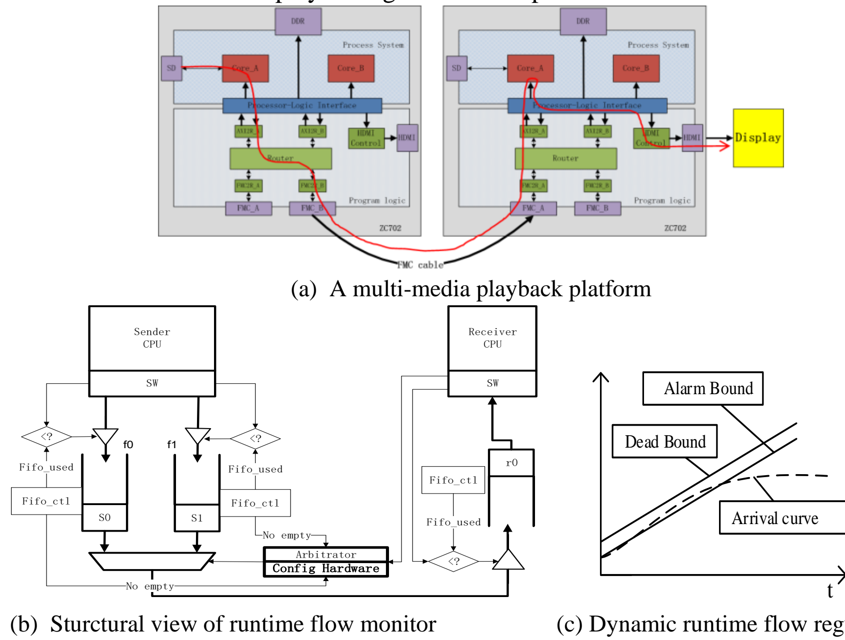
Researches on real-time analysis often focus on design-time (static) analysis of worst-case timing bounds. The validity of the derived bounds should however be monitored and analyzed at runtime to guarantee the system QoS. In our approach, by computing the accurate results of $f\otimes g$, $f\oslash g$, and $f\oslash f$ (arrival curve) at runtime, the hardware architecture can be incorporated in a runtime monitor to ensure that the input flow conforms to its specification and thus to facilitate dynamic QoS fulfillment.

Taking video data stream transfer as an example, we implemented the proposed hardware in a multimedia playback system, as shown in Fig. 7(a). The parameters ($N=128, M=16, C=12$) were chosen by experience. The system is a NoC-based platform using two Xilinx Zynq FPGA evaluation boards (ZC702). Each ZC702 board contains an XC7Z020 SoC and provides peripheral ports including DDR3, HDMI port, SD card and two FMC (FPGA Mezzanine Card) connectors. The XC7Z020 SoC of Xilinx Zynq™-7000 Programmable SoC architecture integrates a dual-core ARM® Cortex™-A9 based Processing System (PS) and Xilinx Programmable Logic (PL).

The two ZC702 boards are connected by an FMC cable. With a router and other interface logic implemented in the PL, the two boards provide a hardware environment for evaluating our design for QoS. In each ZC702 board, the router has four ports and connects two ARM cores and two FMC ports, as shown in Fig. 7(b). The configurable

hardware architecture is used as a runtime flow monitor attached to the arbitrator module for calculating the arrival curve so as to dynamically monitor and shape the input flow.

The prototype is constructed as a client-server system on the two Xilinx FPGA boards. The CPU Core_A in the sender board reads video frame data from the SD card and sends them to the other board (receiver board) through routers and the FMC cable. The software decoder running on the receiver CPU Core_A decodes the video frame data and sends them to the display through the HDMI port.



(a) A multi-media playback platform

(b) Sturctural view of runtime flow monitor

(c) Dynamic runtime flow regulation

Fig. 7. Application to a multi-media playback system

Regarding the arrival curve, we can define two experience-based bounds named Alarm Bound and Dead Bound at design time, as shown in Fig. 7(c). The alarm bound is nearer to the actual arrival curve than the dead bound. Violating the Dead Bound means that data transfers are not valid. Violating the Alarm Bound means that the system should take measures to prevent the possible violation of the dead bound. The arrival curve is calculated by the hardware implementation of our proposed architecture. The comparator of AR and BR is a violating-state indicator whenever a violation occurs.

The advantage of the proposed approach is that it can expose precise details of the behavior of the flow and service: not only if a bound is violated, but also which part violates and how much of violation. Beyond normal functionality, the approach can support finer analysis with more information. For example, when checking how tight the arrival curve bound of the input flow is, the tightest bound curve values from design-time analysis can be defined at each point and be preloaded into the BR registers. When a violation event ($AR(i)>BR(i)$) occurs, it is known that the ith time interval is violated and the volume of violation is calculated by the ith comparator. Such precise information enables the system to react to the violation for precise QoS provisioning.

6 CONCLUSION

To enable application of network calculus to satisfy QoS constraints at runtime, we have for the first time proposed a configurable hardware architecture to realize all essential network calculus operations for processing arrival and service curves. By configuring switches to different data paths, it can calculate arrival curve, min-plus convolution and min-plus deconvolution in a unified pipeline hardware substrate with only one cycle latency. This architecture is implemented and further optimized on an FPGA platform, showing high performance with reasonable resource cost. A case study of a multimedia playback for runtime arrival curve monitoring and QoS has been presented. By enabling to support network calculus operations at a full scale in dynamic environments, this study demonstrates the hardware implementation feasibility of bringing network calculus into action to achieve QoS at runtime beyond what is achievable at design time.

References

1. Rene L. Cruz. *A calculus for network delay, Part I: Network elements in isolation; Part II: Network analysis*. IEEE Trans. on Information Theory, 37(1):114-131, Jan. 1991.
2. J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet* (LNCS 2050). Heidelberg, Germany: Springer, 2004.
3. C.-S. Chang. *Performance Guarantees in Communication Networks*. London, U.K.: Springer-Verlag, 2000.
4. Y. Jiang and Y. Liu. *Stochastic Network Calculus*. London, U.K.: Springer, 2008.
5. Y. Qian, Z. Lu, and W. Dou. *Analysis of worst-case delay bounds for on-chip packet-switching networks*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 29(5):802 –815, 2010.
6. K. Huang, G. Chen, C. Buckl, and A. Knoll. *Conforming the runtime inputs for hard real-time embedded systems*. Proc. of the 49th Design Automation Conference (DAC), 2012.
7. Z. Lu and X. Zhao. *xMAS-Based QoS Analysis Methodology*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(2): 364-377, 2018.
8. B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. *Guaranteed services of the NoC of a manycore processor*. In Proc. of Int. Workshop on Network on Chip Architecture, Cambridge, U.K., 2014, pp. 11–16.
9. E. Wandeler, L. Thiele, et al. *System architecture evaluation using modular performance analysis—A case study*. Software Tools Technology Transfer, vol. 8, pp. 649–667, 2006.
10. Z. Lu, M. Millberg, et al. *Flow Regulation for On-Chip Communication*. Proc. of 2009 Design, Automation and Test in Europe Conference (DATE), Nice, France, April 2009.
11. G. Du, M. Li, et al. *An analytical model for worst-case reorder buffer size of multi-path minimal routing NoCs*. Proc. of Int. Symposium on Networks-on-Chip (NOCS), Sept. 2014.
12. Z. Lu and Yi Wang. *Dynamic Flow Regulation for IP Integration on Network-on-Chip*. The 6th ACM/IEEE International Symposium on Networks-on-Chip (NOCS), May 2012.
13. G. Du, Y. Ou, et al. *OLITS: An Ohm's Law-like traffic splitting model based on congestion prediction*. Proc. of 2016 Design, Automation and Test in Europe Conference, March 2016.
14. Z. Lu and Y. Yao. *Dynamic Traffic Regulation in NoC-Based Systems*. IEEE Trans. VLSI Systems, 25(2): 556-569, 2017.
15. G. Du, G. Liu, et al. *SSS: Self-aware System on Chip using a Static-dynamic Hybrid Method*. ACM Journal on Emerging Technologies in Computing Systems, 15(3):28, April 2019.