RESEARCH-ARTICLE

Optimising the Scheduling of System Level Logical Execution Time Systems

JAMIE LEE, Auckland University of Technology, Auckland, AUK, New Zealand

NATHAN ALLEN, Auckland University of Technology, Auckland, AUK, New Zealand

MATTHEW M Y KUO, Auckland University of Technology, Auckland, AUK, New Zealand

EUGENE YIP

Open Access Support provided by:

Auckland University of Technology

Published: 28 September 2025

Citation in BibTeX format

MEMOCODE '25: International Symposium on Formal Methods and Models for System Design
September 28 - October 3, 2025
Taipei, Taiwan

Conference Sponsors:
SIGDA
SIGBED

https://doi.org/10.1145/3742875.3754684
ISBN: 9798400719943

Optimising the Scheduling of System Level Logical Execution Time Systems

Jamie Lee
Auckland University of Technology
Auckland, New Zealand
jamie.lee@aut.ac.nz

Nathan Allen
Auckland University of Technology
Auckland, New Zealand
nathan.allen@aut.ac.nz

Matthew M.Y. Kuo
Auckland University of Technology
Auckland, New Zealand
matthew.kuo@aut.ac.nz

Eugene Yip
GLIWA GmbH & Co. KG
Weilheim i. OB, Germany
eugene.yip@gliwa.com

Abstract

The paradigm of Logical Execution Time (LET) tasks is widely adopted by major tool vendors for designing deterministic and time-predictable software in multi-core systems, particularly in the automotive industry. To extend the use of LET in distributed environments, System Level Logical Execution Time (SL-LET) has been developed to effectively manage communication and delays between networked devices. However, there is currently a lack of open-source tools available for SL-LET, and the task allocation and scheduling problem for SL-LET remains unsolved.

To address these concerns, we introduces a novel Integer Linear Programming (ILP)-based optimisation approach for SL-LET task allocation and scheduling, focusing on minimising core utilisation and average system response times. To illustrate the effectiveness of the approach, we benchmark our ILP-based solution against a traditional core allocation heuristic across multiple task sets. Through this evaluation, our approach, when compared to the heuristic, is able to demonstrate average response times that are 26.9 % smaller.

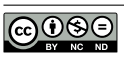
CCS Concepts

• **Software and its engineering** → **System modeling languages**; • **Computer systems organization** → **Real-time languages**; *Distributed architectures*; *Embedded software*; • **Theory of computation** → *Distributed computing models.*

Keywords

Distributed execution platform, real-time programming models, system level logical execution time, integer linear programming

ACM Reference Format:
Jamie Lee, Nathan Allen, Matthew M.Y. Kuo, and Eugene Yip. 2025. Optimising the Scheduling of System Level Logical Execution Time Systems. In *International Symposium on Formal Methods and Models for System Design (MEMOCODE '25), September 28-October 3, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3742875.3754684



MEMOCODE '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1994-3/2025/09
https://doi.org/10.1145/3742875.3754684

1 Introduction

Logical Execution Time (LET) [8] is a paradigm known for its time-deterministic behaviour and consistency in data reading and writing. By specifying periodic time points for reading and writing data, a LET task is ensured to publish data at the intended time and consumed by the correct dependent task instance(s). In fact, the behaviour of a LET system is invariant to the chosen target hardware, deployment strategy, or task scheduling policy. This is attractive for the design and implementation of safety- and time-critical software, and allows for separation of concerns between a system's design and its implementation. For example, the Automotive Open System Architecture (AUTOSAR) standard promotes the use of LET as a way to develop reliable multi-core automotive software.

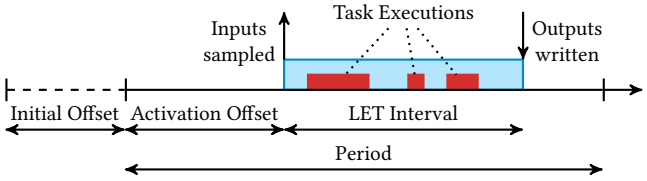


Figure 1: LET task parameters

An example LET task is shown in Figure 1. A LET task has a fixed *period* and *initial offset*. During each period, the task reads all *inputs* and writes all *outputs* at statically determined time points, specifically at the start and end of the *LET interval*. Task execution also occurs within this interval. The *activation offset* shifts the LET interval within the period. These parameters enable the data flow between tasks to be predicted at design time and be invariant to changes in implementation or execution platform.

However, while this is highly desirable for safety- and time-critical applications, there are major drawbacks when it comes to distributed communication. LET assumes that all task communications complete instantaneously, which is unrealistic for distributed systems, or be considered as part of their execution, which incurs scheduling overhead. In such systems, communication between devices incurs network delays due to factors such as data encapsulation, decapsulation, and transmission latency.

System Level Logical Execution Time (SL-LET) has been proposed [4] to address this issue by extending LET to incorporate bounded communication delays into the paradigm. It does so by

decomposing each communication delay into *network* and *protocol* delays, and modelling each as a LET task. These tasks are collectively known as *interconnect tasks*, which can be scheduled like regular LET tasks to ensure that both time and dependency determinism are preserved, even in distributed systems.

However, the research and development of SL-LET systems is non-trivial, and progress in this area relies on tools to validate new concepts and techniques on realistic systems. There are currently no openly available tools for system design using SL-LET that are accessible to researchers. Additionally, the core allocation and scheduling problems for distributed systems, taking into account network delays, present significant challenges in the design of effective and efficient SL-LET systems.

Contributions. We propose a novel Integer Linear Programming (ILP)-based approach to tackle the allocation and scheduling problems in SL-LET systems that aims to produce a schedule with the lowest response times.

2 Related Work

Gemlau et al. utilises the communication paradigm of SL-LET to achieve a proof-of-concept distributed predictable communication [5]. While this work explores the extension of LET and introduces the benefits of SL-LET, optimising end-to-end latency of SL-LET was not considered.

Günzel et al. and Durr et al. acknowledged that for time critical distributed systems, it is essential that tasks meet their correct deadline [3, 6]. Their works present cause-effect chain end-to-end timing analysis by exploring *maximum reaction time* and *maximum data age*. Although Günzel et al. extend the timing analysis to LET, they do not discuss how this could be applied for SL-LET with distributed devices or timing optimisation.

Cause-effect chains for automotive system and its timing analysis and optimisation have been studied for a while now [1, 2, 14, 18]. Data age analysis and optimisation is discussed by Schlatow et al., where a Mixed Integer Linear Program (MILP) approach is used [15]. Each task's offset and priority are put into consideration and the MILP constraints map tasks to processors while the objective function minimises the data age and the response time to ensure that the reduction of data age is prioritised over response time. However, they focus only on multicore system on a single device, and don't directly conform to the LET or SL-LET paradigms. Kordon et al. evaluates the data age of LET by graphing all event chains in the system [10], while Pazzagalia et al. use an MILP approach to manipulate the task-processor mapping for LET tasks and to address *execution interference* [13]. However, they still do not tackle SL-LET over distributed systems.

Reducing the latency for LET is a topic that has previously been investigated in detail [9, 16, 19]. Köhler et al. present the concept of robustness margins for cause-effect chains that ensure end-to-end deadlines are met as long as the software extension is kept within set bounds. The margin is calculated by subtracting a LET instance's duration from its Worst Case Response Time (WCRT), and the paper proves that if the increased response time is less than the robustness margin the deadline will be satisfied. Although the authors formalise robustness margins, they do not discuss optimising the end-to-end latency of task schedules in SL-LET.

Martinez et al. and Wang et al. recognised that, compared to other paradigms such as Dynamic Buffer Protocol (DBP), LET performs poorly in terms of end-to-end latency [11, 17]. To address this limitation, the Flexible Logical Execution Time (fLET) model was proposed and formalised through the introduction of a *virtual offset*, which manipulates a task's release time, and a *virtual deadline*, which allows a task to output early. Günzel and Becker [7] explored the use of task phases to optimise end-to-end latency. However, these approaches do not account for execution across multiple distributed devices as in the SL-LET paradigm.

In summary, while there is a plethora of work on scheduling and event-chain analysis of LET-based systems, until recently very little attention has been paid to their SL-LET counterparts. The addition of distribution and communication delays in SL-LET complicates their end-to-end analysis and scheduling, meaning that traditional LET techniques are not directly applicable in this domain.

3 ILP Constraints

The core allocation and scheduling problem for an SL-LET system *SL* (defined in Appendix A) is formulated using ILP. Here, we restrict numeric definitions to be integer multiples of some arbitrary base timescale, such as 1 ns.

3.1 LET Task Constraints

A set of variables are created for each task instance $t_x^i \in \mathcal{T}_L$ to store the LET interval in terms of its start and end times, denoted $t_x^i.ls$ and $t_x^i.le$ respectively.

$$\forall_{t_x^i \in \mathcal{T}_L}\ t_x^i.ls = t_x.o + t_x.a + i \times t_x.p \tag{1a}$$

$$\forall_{t_x^i \in \mathcal{T}_L}\ t_x^i.le = t_x^i.ls + t_x.\delta \tag{1b}$$

For each task instance t_x^i, its execution time, $t_x^i.s$ to $t_x^i.e$, must be equal to the Worst Case Execution Time (WCET) of the underlying task t_x and within the LET interval.

$$\forall_{t_x^i \in \mathcal{T}_L}\ t_x^i.e - t_x^i.s = t_x.wcet \tag{2a}$$

$$\forall_{t_x^i \in \mathcal{T}_L}\ t_x^i.s \geq t_x^i.ls \tag{2b}$$

$$\forall_{t_x^i \in \mathcal{T}_L}\ t_x^i.e \leq t_x^i.le \tag{2c}$$

3.2 Core Allocation Constraints

Task to core assignments are encoded through a binary variable $a_{x,k} \in A$, where x and k are the task and core indices respectively.

$$\forall_{t_x \in T} \sum_{c_k \in C} a_{x,k} = 1 \tag{3}$$

The possible pairings of task allocations are captured through a binary variable $\psi^{core}_{x,k,y,l}$, which indicates that tasks t_x and t_y are allocated to cores c_k and c_l respectively.

$$\forall_{a_{x,k} \in A}\ \forall_{a_{y,l} \in A}\ \psi^{core}_{x,k,y,l} \leq a_{x,k} \tag{4a}$$

$$\forall_{a_{x,k} \in A}\ \forall_{a_{y,l} \in A}\ \psi^{core}_{x,k,y,l} \leq a_{y,l} \tag{4b}$$

$$\forall_{a_{x,k} \in A}\ \forall_{a_{y,l} \in A}\ \psi^{core}_{x,k,y,l} \geq a_{x,k} + a_{y,l} - 1 \tag{4c}$$

To capture when two tasks t_x and t_y are allocated to different cores, the binary variable $\psi^{task}_{x,y}$ is used.

$$\forall_{t_x \in T}\ \forall_{t_y \in T}\ \psi^{task}_{x,y} = \sum_{c_k \in C} \sum_{\substack{c_l \in C \\ k \neq l}} \psi^{core}_{x,k,y,l} \tag{4d}$$

3.3 Task Execution Constraints

To ensure that the execution for two tasks allocated to the same core does not overlap we introduce a binary variable $b^{task}_{x,i,y,j}$ which indicates that t^i_x occurs after t^j_y. Here, $\mathcal{N}$ is a practically infinite number from the scheduling perspective, where $\mathcal{N} = 2 \times makespan$.

$$\forall_{t^i_x, t^j_y \in \mathcal{T}_L, x \neq y}\ t^i_x.e - t^j_y.s \leq \mathcal{N} \times b^{task}_{x,i,y,j} + \mathcal{N} \times \psi^{task}_{x,y} \tag{5a}$$

$$\forall_{t^i_x, t^j_y \in \mathcal{T}_L, x \neq y}\ t^j_y.e - t^i_x.s \leq \mathcal{N} - \mathcal{N} \times b^{task}_{x,i,y,j} + \mathcal{N} \times \psi^{task}_{x,y} \tag{5b}$$

3.4 Objective 1: Minimise Cores

A binary variable u_k indicates if core $c_k \in C$ has at least one task assigned to it. To minimise the number of cores used, the sum of each of these variables is used.

$$\forall_{a_{x,k} \in A}\ u_k \geq a_{x,k} \tag{6a}$$

$$\forall_{c_k \in C}\ u_k \leq \sum_{t_x \in T} a_{x,k} \tag{6b}$$

Objective: Minimise Cores

$$\min\left(\sum_{c_k \in C} u_k\right) \tag{7}$$

3.5 Objective 2: Minimise Response Time

Two variables, $\lambda_{k,l}$ and $\lambda_{x,y}$, contain the communication delays between cores (c_k to c_l) and tasks (t_x to t_y) respectively.

$$\lambda_{k,l} = \begin{cases} 0, & \text{if } \mathrm{dev}(k) = \mathrm{dev}(l) \\ link.encap + link.net + link.decap, & \\ \quad \text{if } \exists link \in Link : link.src = \mathrm{dev}(k) \wedge link.dest = \mathrm{dev}(l) & \\ \mathcal{N}, & \text{otherwise} \end{cases} \tag{8a}$$

$$\lambda_{x,y} = \sum_{c_k \in C} \left(\sum_{c_l \in C} \psi^{core}_{x,k,y,l} \times \lambda_{k,l} \right) \tag{8b}$$

To handle initial dependencies, each task t_x also has a *bottom* instance $t^\perp_x$ some large time in the past.

$$\forall_{t_x \in T}\ t^\perp_x.ls = -\mathcal{N} \tag{8c}$$

$$\forall_{t_x \in T}\ t^\perp_x.le = t^\perp_x.ls + t_x.\delta \tag{8d}$$

To determine dependencies between task instances, a binary variable $b^{dep}_{x,i,y,j}$ is used to indicate whether t^j_y depends on t^i_x. There can be only one source for each dependency, and the source's end time must allow for the communication delay $\lambda_{x,y}$ to be handled.

$$\forall_{d \in D}\ \forall_{t^i_x \in t_{d.src} \cup t^\perp_{d.src}}\ \forall_{t^j_y \in t_{d.dest}}\ t^i_x.le + \lambda_{x,y} - t^j_y.ls \leq \mathcal{N} - \mathcal{N} \times b^{dep}_{x,i,y,j} \tag{8e}$$

$$\forall_{d \in D}\ \forall_{t^j_y \in t_{d.dest}} \sum_{t^i_x \in t_{d.src} \cup t^\perp_{d.src}} b^{dep}_{x,i,y,j} = 1 \tag{8f}$$

Delays are calculated for each task instance pair through a variable $delay_{x,i,y,j}$, using $b^{dep}_{x,i,y,j}$ to determine whether a dependency exists between t^i_x and t^j_y. The sum of these delays is then minimised in order to reduce the average response time.

$$\forall_{t^i_x \in \mathcal{T}_L}\ \forall_{t^j_y \in \mathcal{T}_L}\ delay_{x,i,y,j} \geq t^j_y.ls - t^i_x.le - \mathcal{N} + \mathcal{N} \times b^{dep}_{x,i,y,j} \tag{9a}$$

$$\forall_{t^i_x \in \mathcal{T}_L}\ \forall_{t^j_y \in \mathcal{T}_L}\ delay_{x,i,y,j} \leq t^j_y.ls - t^i_x.le + \mathcal{N} - \mathcal{N} \times b^{dep}_{x,i,y,j} \tag{9b}$$

$$\forall_{t^i_x \in \mathcal{T}_L}\ \forall_{t^j_y \in \mathcal{T}_L}\ delay_{x,i,y,j} \geq 0 \tag{9c}$$

Objective: Minimise Response Time

$$\min\left(\sum_{t^i_x \in \mathcal{T}_L, t^j_y \in \mathcal{T}_L} delay_{x,i,y,j} \right) \tag{10}$$

4 Evaluation

In order to evaluate the proposed approach, a number of experiments were performed across a range of task sets. The two ILP-based algorithms, along with a simple greedy heuristic that allocates tasks to the least-utilised core, are executed over the same task sets, allowing us to compare their performance and outputs.

All algorithms and associated code were implemented in Python using the library PuLP [12] and the ILP solver Gurobi. Variables and constraints were created according to the formalisations in Section 3. The code was executed using Python 3.13 on an Intel Core i9-9900k @ 3.60 GHz with 32 GB of RAM running Ubuntu 22.04.5 LTS. A timeout of 5 min (300 s) was set in Gurobi.

Task sets were created over an increasing number of tasks (starting from 1) in order to evaluate the scalability of the algorithms. At each size, we randomly create 10 schedulable task sets. For each task a random period, initial offset, LET duration, and WCET are chosen. The period is chosen from a set of possible values (1 ms, 2 ms, 5 ms, 10 ms, 20 ms) in order to simplify the Least Common Multiple (LCM) as part of the hyperperiod calculation. Values are chosen for the initial offset (up to 2 ms), LET duration (up to the selected period), and WCET (up to 1 ms). Additionally, a total of $2 \times (N - 1)$ dependencies are generated between the N tasks.

The evaluation was run over a distributed physical system configuration in order to illustrate the applicability of this approach for multi-device allocation problems. The system contains 3 cores spread over 2 devices, each with a protocol delay of 0.5 ms, connected via a network link of 1 ms.

4.1 Experimental Results

Figures 2 and 3 show the performance of the various approaches in terms of the optimisation metrics that we are looking at — number of cores used and average dependency delay. Firstly, in terms of core usage (Figure 2) we can see that the Minimise Cores (MC) ILP approach always uses fewer or equal cores to the Minimise Response Time (MRT) approach, averaging 0.13 (5.4 %) fewer cores. On the other hand, the weaknesses of the heuristic are immediately clear, as it aims to utilise all possible cores.

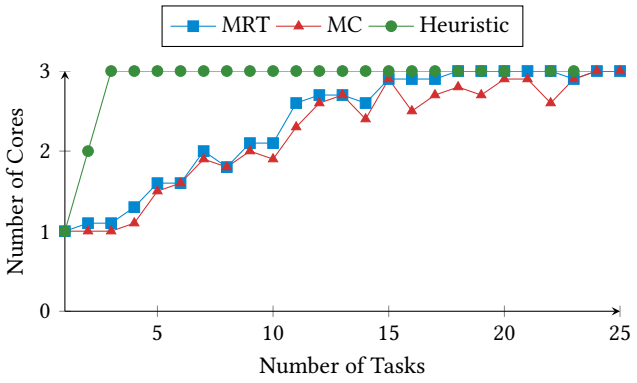


Figure 2: Average number of cores allocated

For average dependency delay (Figure 3 we can again see a similar, but opposite, trend with the ILP approaches, where the MRT approach always has the minimal delay and the MC approach averages 0.55 ms (13.2 %) longer. Meanwhile, the heuristic also seems to be in the similar ballpark, with it sometimes matching the MRT ILP and generating delays that are, on average, 1.15 ms (26.9 %) longer.

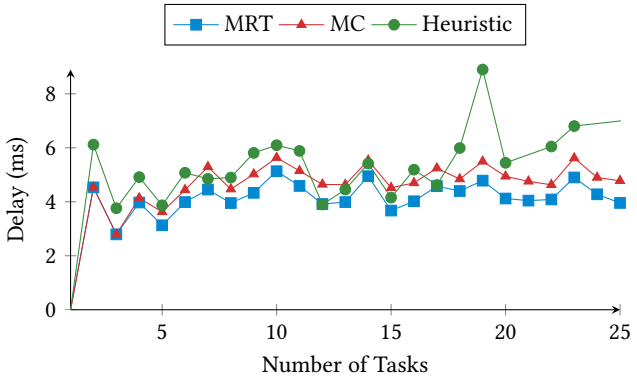


Figure 3: Average dependency delay

Next, we also plot the spread of cores used against overall system utilisation in Figure 4, which nicely illustrates the differences between the various approaches. Firstly, the behaviour of the heuristic approach is apparent, with it using all available cores. Similarly, for the ILP approaches we can see that the MRT approach is more likely to use additional cores, even at lower system utilisations.

Finally, to evaluate scalability, Figure 5 shows the time required to solve the allocation problem as the number of tasks increases.

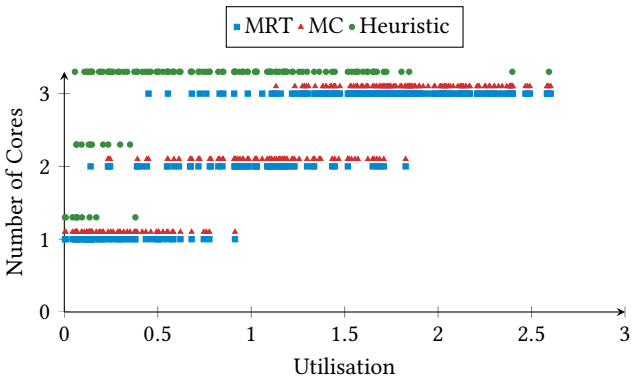


Figure 4: Number of cores used based on system utilisation

Here, we see the benefit of the heuristic approach which requires minimal solving time. Between the two ILP-based approaches, we see exponential growth in the solve time, as expected, with the MRT approach increasing more rapidly than that of the MC.

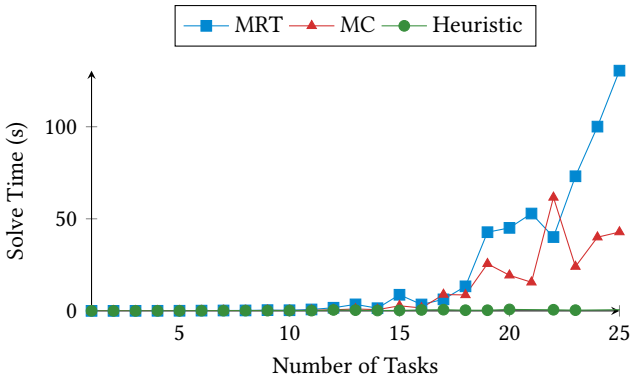


Figure 5: Average solve time

5 Conclusions

Independently addressing the task allocation and scheduling may result in inefficient or unschedulable SL-LET systems. To address this, we proposed an ILP-based technique to tackle both problems simultaneously. Two optimisation objectives were defined to either minimise the communication delay between tasks in the system, or minimise the number of cores used (like a packing problem).

To illustrate the effectiveness of this approach, we performed a series of evaluations on randomly generated task sets over a distributed system. The ILP approaches were compared with a simple heuristic, which tackles the allocation and scheduling problems separately, to show the importance of solving them together. On average, the ILP approaches produce inter-task delays that are 26.9 % smaller and core usage that is 37.5 % lower than the heuristic.

In the future, we aim to investigate several extensions to this work, such as heterogeneous computing architectures or adding support for aperiodic tasks. Additionally, while this work optimises for the *average case*, it does not minimise the *worst case*, which would require evaluating event chains.

References

[1] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2017. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture* 80 (2017), 104–113. https://doi.org/10.1016/j.sysarc.2017.09.004

[2] Matthias Becker and Saad Mubeen. 2018. Timing Analysis Driven Design-Space Exploration of Cause-Effect Chains in Automotive Systems. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, New York City, USA, 4090–4095. https://doi.org/10.1109/IECON.2018.8592842

[3] Marco Dürr, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. 2019. End-to-End Timing Analysis of Sporadic Cause-Effect Chains in Distributed Systems. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 58 (Oct. 2019), 24 pages. https://doi.org/10.1145/3358181

[4] Kai-Björn Gemlau, Hermann v. Hasseln, and Rolf Ernst. 2022. Industry-track: System-Level Logical Execution Time for Automotive Software Development. In *2022 International Conference on Embedded Software (EMSOFT)*. IEEE, New York City, USA, 21–23. https://doi.org/10.1109/EMSOFT55006.2022.00017

[5] Kai-Björn Gemlau, Leonie Köhler, and Rolf Ernst. 2021. Efficient Run-Time Environments for System-Level LET Programming. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, New York City, USA, 749–754. https://doi.org/10.23919/DATE51398.2021.9474257

[6] Mario Günzel, Harun Teper, Georg von der Brüggen, and Jian-Jia Chen. 2024. End-To-End Latency of Cause-Effect Chains: A Tutorial. *ACM Trans. Embed. Comput. Syst.* 24, 1, Article 22 (Dec. 2024), 18 pages. https://doi.org/10.1145/3703630

[7] Mario Günzel and Matthias Becker. 2025. Optimal Task Phasing for End-To-End Latency in Harmonic and Semi-Harmonic Automotive Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (preprint)*. IEEE, New York City, USA, 13 pages.

[8] Christoph M. Kirsch and Ana Sokolova. 2012. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems*. Springer, Berlin, Heidelberg, 103–120.

[9] Leonie Köhler, Phil Hertha, Matthias Beckert, Alex Bendrick, and Rolf Ernst. 2023. Robust Cause-Effect Chains with Bounded Execution Time and System-Level Logical Execution Time. *ACM Trans. Embed. Comput. Syst.* 22, 3, Article 50 (April 2023), 28 pages. https://doi.org/10.1145/3573388

[10] Alix Munier Kordon and Ning Tang. 2020. Evaluation of the Age Latency of a Real-Time Communicating System using the LET paradigm. In *ECRTS 2020 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*, Marcus Voelp (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Modena, Italy, 20:1–20:20. https://doi.org/10.4230/LIPIcs.ECRTS.2020.20

[11] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. 2018. Analytical Characterization of End-to-End Communication Delays With Logical Execution Time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2244–2254. https://doi.org/10.1109/TCAD.2018.2857398

[12] Stuart Mitchell, Michael OSullivan, and Iain Dunning. 2011. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand* 65 (2011), 25.

[13] Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. 2019. Optimizing the Functional Deployment on Multicore Platforms with Logical Execution Time. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, New York City, USA, 207–219. https://doi.org/10.1109/RTSS46320.2019.00028

[14] A. C. Rajeev, Swarup Mohalik, Manoj G. Dixit, Devesh B. Chokshi, and S. Ramesh. 2010. Schedulability and end-to-end latency in distributed ECU networks: formal modeling and precise estimation. In *Proceedings of the Tenth ACM International Conference on Embedded Software* (Scottsdale, Arizona, USA) *(EMSOFT '10)*. Association for Computing Machinery, New York, NY, USA, 129–138. https://doi.org/10.1145/1879021.1879039

[15] Johannes Schlatow, Mischa Mostl, Sebastian Tobuschat, Tasuku Ishigooka, and Rolf Ernst. 2018. Data-Age Analysis and Optimisation for Cause-Effect Chains in Automotive Control Systems. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. IEEE, New York City, USA, 1–9. https://doi.org/10.1109/SIES.2018.8442077

[16] Yue Tang, Xu Jiang, Nan Guan, Dong Ji, Xiantong Luo, and Wang Yi. 2023. Comparing Communication Paradigms in Cause-Effect Chains. *IEEE Trans. Comput.* 72, 1 (2023), 82–96. https://doi.org/10.1109/TC.2022.3197082

[17] Sen Wang, Dong Li, Ashrarul H. Sifat, Shao-Yu Huang, Xuanliang Deng, Changhee Jung, Ryan Williams, and Haibo Zeng. 2024. Optimizing Logical Execution Time Model for Both Determinism and Low Latency. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, New York City, USA, 135–148. https://doi.org/10.1109/RTAS61025.2024.00019

[18] Shumo Wang, Yuhan Lin, Zhiwei Feng, Maoyang Shan, Qingxu Deng, and Zonghua Gu. 2024. *End-to-End Timing Analysis of Task Chains for TSN-Based Distributed Real-Time Systems Based on IEEE 802.1Qcr.* Technical Report. Northeastern University, Shengyang, China. https://doi.org/10.2139/ssrn.5096121

[19] Risheng Xu, Marvin Kühl, Hermann Von Hasseln, and Dirk Nowotka. 2023. Reducing Overall Path Latency in Automotive Logical Execution Time Scheduling via Reinforcement Learning. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems* (Dortmund, Germany) *(RTNS '23)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/3575757.3593658

A SL-LET Formalisation

Definition A.1 (System Level Logical Execution Time System). An SL-LET system is a tuple $SL = (P, L)$, where:

- P is the physical system (Definition A.2).
- L is the underlying LET system (Definition A.3).

Definition A.2 (Physical System). A physical system is a tuple $P = (Dev, Link)$, where:

- Dev is a set of devices with each device $dev \in Dev$ being a singleton (C) where C is a set of availabe processing cores.
- $Link$ is a set of links between devices with each link $link \in Link$ being a tuple $(src, dest, encap, decap, net)$, where:
  - $src \in Dev$ is the source device,
  - $dest \in Dev$ is the destination device,
  - $encap \in \mathbb{N}$ is the protocol delay of src,
  - $decap \in \mathbb{N}$ is the protocol delay of $dest$, and
  - $net \in \mathbb{N}$ is network delay between src and $dest$.

Definition A.3 (Logical Execution Time System). A LET system is a tuple $L = (T, D, \Gamma)$, where:

- T is the set of all LET tasks in the system with each task $t \in T$ being a tuple $(p, o, a, \delta, wcet)$, where:
  - $p \in \mathbb{N}$ is its period.
  - $o \in \mathbb{N}$ is its initial offset.
  - $a \in \mathbb{N}$ is its activation offset.
  - $\delta \in \mathbb{N}$ is the duration of its LET interval.
  - $wcet \in \mathbb{N}$ is its WCET.
- D is a set of communication dependencies with each dependency $d \in D$ being a tuple $(src, dest)$, where:
  - $src \in T$ is the source task.
  - $dest \in T$ is the destination task.
- Γ is a set of all event chains with each event chain $\gamma \in \Gamma$ being a sequence $\langle d_0, d_1, \cdots, d_{n-1} \rangle$ of dependencies.
  - $\forall_{i \in \{1,\ldots,|\gamma|-1\}} d_{i-1}.dest = d_i.src$.

For each task $t_x \in T$, t_x^i denotes the ith instance of its execution. Let $\mathcal{T}_{t_x} = \left\{ t_x^i \mid t_x.o + i \times t_x.p < makespan, i \in \mathbb{N}_0 \right\}$ be the set of all instances of task t_x, and let $\mathcal{T}_L = \bigcup_{t_x \in T} \mathcal{T}_{t_x}$ be the set of all instances of all tasks in the LET system L.

For a given LET system L, the hyperperiod (HP) is defined as the LCM of all task periods (eq. (11a)) and the "hyperoffset" (HO) as the maximum of all task offsets (eq. (11b)). For a given physical system P, the "hyperdelay" (HD) is defined as the maximum of all link delays (eq. (11c)). The minimum makespan required to ensure that all dependencies are satisfied at least once is shown in eq. (11d).

$$HP = \text{lcm}\left(\{t_x.p \mid t_x \in T\}\right) \tag{11a}$$

$$HO = \max\left(\{t_x.o \mid t_x \in T\}\right) \tag{11b}$$

$$HD = \max\left(\{link.encap + link.net + link.decap \mid link \in Link\}\right) \tag{11c}$$

$$makespan \geq \left(2 + \left\lceil \frac{HD}{HP} \right\rceil\right) \times HP + HO \tag{11d}$$