Proceedings of the

17th Junior Researcher Workshop on Real-Time Computing

JRWRTC 2024

Porto, Portugal

November 6, 2024

Message from the Workshop Chairs

Welcome to the 17th Junior Researcher Workshop on Real-Time Computing, held in conjunction with the 32nd International Conference on Real-Time Networks and Systems (RTNS) in November 2024. The workshop provides an informal environment for junior researchers, where they can present their ongoing work in a relaxed forum and engage in enriching discussions with other members of the real-time systems community.

We are grateful to the Program Committee for their thorough evaulations of and feedback for the submitted papers. We would also like to thank the authors who submitted to the conference; its continued success as a forum to encourage junior researchers in the community is due to your efforts, authors, and we wish you success as you continue your careers. This year, the workshop accepted five peer-reviewed papers.

We hope that you all enjoy the workshop, and take the opportunity to engage with our junior colleagues in discussing their work during the poster session and beyond.

Tanya Amert – Carleton College, Northfield, MN, USA

Federico Aromolo – Scuola Superiore Sant'Anna, Pisa, Italy

JRWRTC 2024 Workshop Chairs

Program Committee

Shareef Ahmed – University of North Carolina at Chapel Hill, USA

Hyunjong Choi – San Diego State University, USA

Anaïs Finzi – TTTech Computertechnik AG, Austria

Gautam Gala – TU Kaiserslautern-Landau, Germany

Miguel Gutiérrez Gaitán – Pontificia Universidad Católica de Chile, Chile

Woosung Kang – DGIST, South Korea

Biruk Seyoum – Columbia University, USA

Marion Sudvarg – Washington University in St. Louis, USA

Binqi Sun – TU Munich, Germany

Harun Teper – TU Dortmund, Germany

Raffaele Zippo – University of Pisa, Italy

Contents

Proceedings edited by the JRWRTC 2024 Workshop Chairs - Tanya Amert and Federico Aromolo.

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.

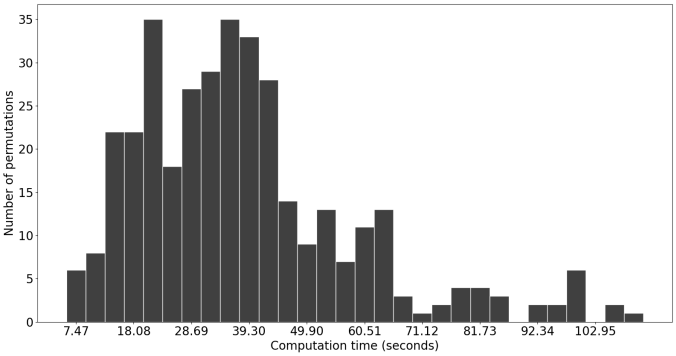


Fig. 1. Distribution of runtimes to compute the (min,+) convolution of a set of 6 curves, according to all possible permutations of the set. Minimum value: 5.7s; Maximum value: 111.8s; Average: 38.48s.

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

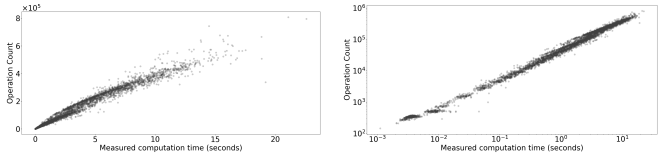


Fig. 2. Graphs showing measured time (x-axis) and prediction using *Operation Count* (y-axis) on both a linear (left) and a logarithmic (right) scale.

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*.

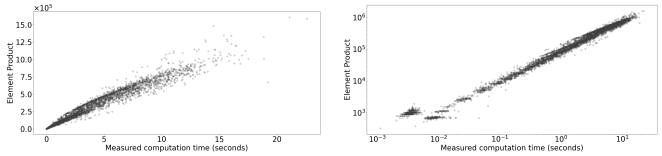


Fig. 3. Graphs showing measured time (x-axis) and prediction using *Element Product* (y-axis) on both a linear (left) and a logarithmic (right) scale.

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

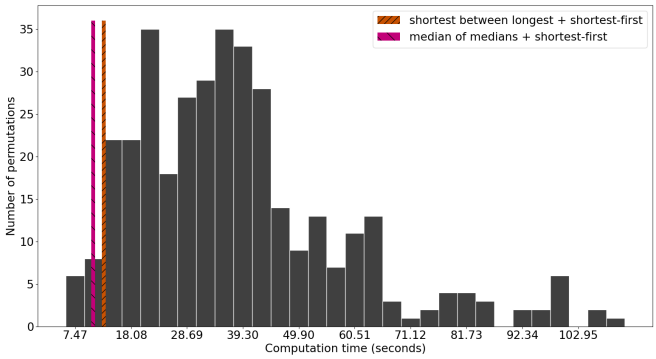


Fig. 4. Distribution of runtimes to compute the (min,+) convolution of a set of 6 curves, according to all possible permutations of the set, vs. the runtimes obtained using our heuristics.

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.

Overview of the Real Time DPU prototype for space instrument DUSTER

Juan Manuel Gómez López
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Cientificas
Granada, Spain
orcid.org/0000-0001-6295-4035

M. Carmen Pastor Morales
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/0000-0002-3877-4277

Francisco Lobón Villanueva
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/0009-0001-5311-5499

Gian Paolo Candini
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/0000-0001-9481-8206

Rosario Sanz Mesa
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/ 0000-0002-7109-9349

Julio F. Rodríguez Gómez
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/0000-0002-6757-5912

Miguel A. Sanchez Carrasco
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain
orcid.org/0000-0001-5533-3660

Nicolas Robles
Instituto de Astrofísica de Andalucía
Consejo Superior de Investigaciones Científicas
Granada, Spain

***Abstract*— This paper proposes an architecture for the data processing unit of the DUSTER instrument for exploration missions to solar system bodies. The DUSTER project objective is to develop instrumentation and technologies for in situ analysis of the charging and cohesion of the dust grain in the oon regolith of the Moon. There are three probes: one shall measure the electric field generated by the dust particles; the second measures the status of the plasma in the surrounding near the regolith and the third analyses the movement of the grains measuring the current resulting from the impact of the charged grains onto the electrodes with high voltage (thousands) applied.**

The DUSTER project is funded by the European Commission with BIRA-IASB leading the project and IAA-CSIC, ONERA and THALES as partners. IAA-CSIC is developing the electronics required to acquire in real time the data from the probes, process it and send it back to Earth for post-analysis. We approach to the development of the instrument from the point of view of a model-based design to address the complexity associated with development of space projects.

The DUSTER hardware platform, prototyped on a Field Programmable Gate Array and relying on the GRLIB IP Library, is designed such that it can be a candidate for qualification and use in future ESA, NASA or JAXA missions, which cannot rely on high performance COTS technologies. In particular, the platform is targeted to be a mixed-criticality platform, allowing the deployment of the instrument in different levels of criticality.

Keywords—DUSTER, FPGA, LEON3

I. INTRODUCTION

A. *DUSTER*

International scientific and commercial interests in exploration missions to solar system bodies such as the Moon, asteroids and comets have increased significantly during the last two decades and will continue to increase in the future. One major environmental constraint during exploration missions is the presence of charged dust-like particles that can be a threat for both human and robotic exploration missions.

The DUSTER (for DUst Study, Transport, and Electrostatic Removal for exploration missions) project aims to analyse the effects of dust adhesion to any type of equipment (e.g. astronauts, rovers, landers and scientific equipment). DUSTER is part of a UE funded project [1] with BIRA-IASB [1] leading the project and TAS-E [2] (THALES Spain), ONERA[3] and IAA-CSIC[4] as partners. BIRA-IASB is coordinating the project and developing the instrument front-ends and provide the high-power supplies of the instrument. IAA-CSIC is in charge of developing the tools (software and electronics) required to acquire the data, process it and send it back to Earth for post-analysis (Electrical Ground Support Equipment, EGSE, data processing unit, DPU, breadboard architecture and the low-voltage power supplies of the instrument). TASE will run EMC tests of the integrated instrument. ONERA is in charge of developing the system (around Technology Readiness Level 4, TRL4) that will charge the dust, activate their transport and measure the generated current and thus the net charge.

The DUSTER instrument shall answer the key questions: what voltage and what type of electrodes are needed to remove the dust depending on the dust properties and the environment (UV, plasma, E-field). The DUSTER instrument uses the High- and Low- voltage power supplies to transport the dust to the probes. Then it detects the current carried by dust grains when they collide with an electrode equipped with high sensitive current amplifiers. From the literature, the charge carried by a single grain is in the order of a few fC up to a few tens of fC and can be mobilized over a few (tens of) centimetres ([2], [3], [4], [5]). Dust charge monitoring is performed with a charge amplifier connected to the output of a Faraday cup (FC). When a dust grain enters the FC, its charge generates a displacement current in the order of the carried charge divided by the dust velocity. The dust velocity itself depends on the dust charge, on the electric field and on the acceleration zone.

B. *Instrument Description*

The DUSTER instrument measures the dust properties and the environment (ultraviolet, plasma, electric field) with three probes: E-Field, DUST and Langmuir. The Low-Voltage

[1] Royal Belgian Institute for Space Aeronomy
[2] THALES Alenia Space Spain
[3] French Aeroespace Lab
[4] Instituto de Astrofísica de Andalucía

Funded by the European Union's HORIZON Research and Innovation programme under grant agreement No 101082466.

Power Supply (LVPS) and the High-Voltage Power Supply (HVPS) generate an electric field that transport the dust to the instrument probes. The data processing unit (DPU) manages the probes and the power supplies to perform the acquisition, process the acquired data and generates science and housekeeping telemetries.

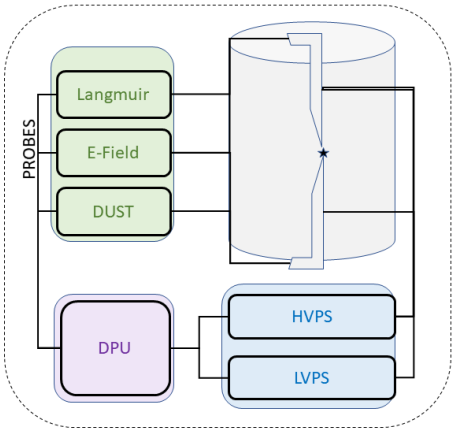


Figure 1 DUSTER Instrument block diagram

Each of the probes has its own front-end that abstracts the management of the analog to digital converter (ADC) and is configured using a serial peripheral interface (SPI).

- The Langmuir probe acquires the current measured by the probe when it operates in sweep, fixed bias and current monitoring mode, as well as the potential of the probe (it monitors the applied bias) in bias monitoring mode.
- The E-Field probe measures the electric field in nominal mode and the voltage provided by the digital to analog converter (DAC) in monitoring mode.
- The Dust probe measures the current acquired by the probe in nominal mode, the current provided by the DAC in current monitoring mode and the voltage applied to the probe (by the HVPS and the ADC) in potential monitoring mode.

The high-voltage power supply (HVPS) receives commands and transmits housekeeping information to the DPU via a serial interface. The output voltage of the HVPS, which is around 5kV, is configurable and shall be adapted to the operation of the DUST probe.

The low-voltage power supply (LVPS) powers the digital components of the instrument and generates the mid-range and low voltages required for the transportation of the dust particles.

The data processing unit (DPU) controls the operation of the probes and the power supplies to perform the science operation, acquires the measurements obtained by the probes, processes them and transmits the results. Moreover, the DPU receives commands with the operation parameters of the instrument.

C. Data production and handling

The operation of the instrument involves transporting dust particles in vacuum environment to the probes and measuring their properties upon collision with a grid. The charged dust particles are attracted to the grid by generating an electric field using the LVPS and HVPS. The instrument will monitor dust counts over several tens of seconds, acquiring data at the highest possible rate to effectively distinguish between consecutive dust particles.

Table 1 Acquisition units for the DUSTER probes

Probe	ADC Sampling (max-min)	DAC	SPI Baudrate
Langmuir	(5kHz-1kHz) / 20 bits	100Hz/16bits	10 - 1 Mb/s
E-Field	1 kHz / 20 bits	1kHz/16 bits	10 - 1 Mb/s
DUST	(10 kHz-1kHz) / 20 bits	1kHz/16 bits	750 – 300 Kb/s

The requirements include both an objective and minimum performance for the acquisition quantities. The allowed data rate for SPI is between 1Mbps and 10 Mbps for the SLP and E-field probes. The DUST probe, because of the high-voltage isolation, data rate is constrained to 300 kbps and 750 kbps. The Langmuir probe must perform bias sweeps with at least 600 steps per sweep and a time resolution of 1 minute, with a goal of achieving 1000 steps per sweep. The E-Field probe will conduct measurements with a time resolution of less than 1 minute. Lastly, dust probe measurements must have a time resolution of at least 10 milliseconds. Table 1 presents the expected acquisition rates of the probes.

II. DUSTER DPU Overview

The DUSTER DPU will execute all software related to instrument management and status monitoring, generating housekeeping data that reflects the instrument's status. It will receive telecommands to adjust the operational parameters of the instrument. The real-time data acquisition task involves the DPU continuously reading the ADC measurements for at least 1 minute. The scientific output of the DUSTER DPU will be a timestamped time series of measurements for both the ambient environment and the dust collected by the probes.

Space systems cannot utilize commercial of the shelf (COTS) processors because they are not designed to support radiation and do not meet qualification requirements. Current space missions often incorporate powerful radiation-hardened ASIC processors[6][7], such as the GR712RC and GR740 from Frontgrade Gaisler. However, while these ASICs include the most common interfaces used in space missions, they lack the flexibility needed to adapt to the specific requirements of the DUSTER project.

In contrast, the soft processors LEON3[8] and NOEL-V[9] are implemented as synthesizable soft intellectual property (IP) cores, allowing for customization and deployment in field-programmable gate arrays (FPGAs). The LEON3 processor, in particular, has been successfully used in various space missions, demonstrating its effectiveness and reliability in harsh environments[10].

The DUSTER DPU is a dual-core system based on the LEON3 processor, which adheres to the SPARC v8 architecture and is built using the GRLIB GPL v2023.2[11]. The GRLIB IP Library, developed by Frontgrade Gaisler, is a comprehensive suite of reusable IP cores designed for system-on-chip (SoC) development. Its primary infrastructure is available as open-source under the GNU GPL license, promoting accessibility and innovation.

A notable feature of the LEON3 processor is its fault tolerance capability against Single Event Upsets (SEUs), particularly in its radiation hardened version, LEON3FT

(Leon3 Fault Tolerant). This fault tolerance mechanism is primarily aimed at protecting on-chip RAM blocks used for IU/FPU register files and cache memory. However, it's important to note that some cores and configurations are exclusive to the commercial version of GRLIB, and the LEON3FT is only available in the fault tolerant (FT) and FT-FPGA versions.

One key advantage of the GRLIB IP Library is its vendor independence, allowing compatibility with various CAD tools and target technologies. This flexibility facilitates seamless integration within model-based development workflows, where projects are constructed using models to represent subsystems and elements rather than relying solely on traditional documentation.

In summary, the DUSTER DPU takes advantage of the capabilities of the LEON3 processor and the versatile GRLIB IP Library, provides a solid foundation for advanced SoC development focused on reliability and integration.

III. Architectural Configuration

The DUSTER instrument is currently under development and therefore changes to the current configuration are expected and will have to be adapted to changing requirements as they arise during development. The development of the different elements of the instrument is being done in parallel and several releases have been scheduled for integration and verification. Once the software and the probes are integrated and the instrument is fully functional, the final platform configuration could be determined.

The baseline configuration of the platform consists of a dual-core Leon3 processor with a system clock of 100MHz. Each CPU features private L1 caches for instructions and data, with 16KB size and 16 bytes cache line length. The L2 cache is disabled. The processor is connected to the debug support unit which is accessible using the JTAG (Joint Test Action Group) and UART (Universal Asynchronous Receiver Transmitter) debug links.

The design includes two memory controllers, one to manage the non-volatile flash memory and the other for the DDR4 volatile memory devices included in the platform. The flash memory is used as permanent storage for the software and firmware of the FPGA. The DDR4 memory is used as the main processor memory for the software execution. Additionally, communication interfaces such as SPI and UART facilitate data exchange with the probes and communication with the control unit. Finally, the Timer Units, Interrupt Controller and General Purpose Input Output (GPIO) controller peripherals, are included in the design to enhance functionality and performance of the system. All these peripherals will be implemented as IP Cores and are provided by the GRLIB IP library. The block diagram of the DUSTER DPU with its main elements is represented in Figure 2.

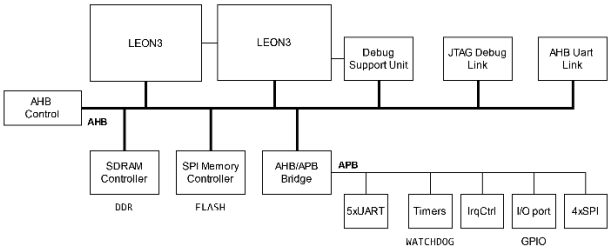


Figure 2 DUSTER DPU block diagram

The SPI interfaces facilitate communication with the probes, with three independent SPI interfaces in the instrument—one for each probe. This design ensures that each probe could meet its timing constraints. The DAC of the probe is also configured using this interface.

Additionally, two UART interfaces connect to the Low-Voltage and High-Voltage power supplies. These interfaces are essential for configuring the power supply outputs, thereby generating the electric fields necessary for observation.

The DPU receives telecommands through one of the UART interfaces, which includes configuration parameters for the instrument's operation. This same interface is also used to transmit housekeeping data and telemetry related to the instrument's products.

Finally, there is an additional UART interface dedicated to logging purposes.

IV. FPGA Device Trade off

The DUSTER DPU will be prototyped on an FPGA, targeting an architecture suitable for future institutional space missions. To develop the prototype, a large FPGA is necessary to accommodate a multicore system along with the required peripherals and numerous I/O ports. Table 2 summarizes the hardware resources of the most common space FPGAs.

Table 2 Rad-Hard FPGA resources

	LUTs (k) / Flip Flops (k)	**I/O ports**
Xilinx XQRKU060	331/663	620
NanoXplore NG-LARGE	137/129	740
Microchip Axcelerator	21/21	198
Microchip RTG4	151/151	720

We opted not to consider RadHard FPGAs due to their higher cost and the project's low Technology Readiness Level (TRL4). However, we aim to stay as close to RadHard specifications as possible.

The Xilinx XQRKU060 FPGA[12], radiation testing has confirmed that the FPGA is suitable for all orbits, including deep space exploration. The QMLB and QMLY-grade version of the XQRKU60 device could be procured for the final hardware prototype. Furthermore, the Xilinx Kintex UltraScale family[13] includes the XCKU060 FPGA, which is a commercial-grade FPGA, but can serve as a basis for understanding the architecture and capabilities of the XQRKU060. Table 3 presents the hardware resources of the Xilinx UltraScale devices.

Table 3 Xilinx UltraScale family resources

	XCKU040	XCKU060	XQRKU060
Radiation Hardness	None	None	Tolerant
System Logic Cells (K)	530	726	726
CLB Flip-Flops (K)	448	663	663
CLB LUTs (K)	242	331	331
DSP Slices	1920	2760	2760
Block RAM (Mb)	21.1	38.0	38.0
Gb/s Transceivers	20	32	32
I/O Pins	520	624	620

The use of commercial-grade FPGAs offers several advantages, including availability of development boards like the GR-VPX-XCKU060 and GR-CPCIS-XCKU from Frontgrade Gaisler, as well as the ADA-SDEV-KIT3 from Alpha Data. In addition, the Xilinx KCU105 Evaluation Kit, featuring the Xilinx Kintex UltraScale XCKU040 FPGA, provides a robust platform for development.

Additionally, from a firmware development perspective, prototyping for any device in the Xilinx Kintex UltraScale family is effectively the same as prototyping for the XCKU060. The Kintex UltraScale family shares a common architecture, meaning that the underlying hardware resources—such as logic cells, DSP blocks, and memory interfaces—are similar across devices. This consistency allows the usage of a full commercial development board, simplifying procurement and reducing costs.

Prototyping on the Kintex UltraScale XCKU040 device provides a reliable reference for the XQRKU060. The Xilinx KCU105 Evaluation Kit allows to prepare the testing, validation and verification at early stages of the project.

Furthermore, the broader ecosystem surrounding the Kintex UltraScale family, including community resources, forums, and documentation, provides extensive support for developers. This shared knowledge base can be leveraged across different devices, further simplifying the development process.

V. SOFTWARE SUPPORT

The LEON3 architecture is supported by the real-time operating system RTEMS [14] which is pre-qualified for space applications[15]. The prequalification toolkit allows end-users to qualify their space applications. That reduces both the cost and the effort of the development and qualification.

VI. SYNTHESIS RESULTS

The initial step of this DUSTER project involves synthesizing the design shown in Figure 2, followed by benchmarking and analysing its compliance with real-time requirements of the instrument.

We prototyped for the Xilinx KCU105 Evaluation Kit for this synthesis and the actual occupations of the FPGA in terms of slice registers (mostly Flip Flops), combinational logic blocks (CLBs) and the global numbers of occupied I/O is presented in Table 4.

Table 4 Device Utilization Summary.

KCU105 Board	**Occupied resources**	
CLB	9373 of 30300	30%
LUT as Memory	1805 of 112800	1.60%
LUT Flip Flop	15227 of 242400	6.28%
Block RAM	74 of 600	12.33%
DSPs	11 of 1920	0.57%
I/O	221 of 520	40%
GLB CLK BUFF	13 of 480	2.71%

Considering the current state of the project development, the resource utilization of the FPGA design seems adequate, enabling us to continue advancing without concern that any new requirements may render development on this platform unfeasible.

VII. RESULTS AND DEVELOPMENT STATUS

In this short paper, we describe the development of the Data Processing Unit for the DUSTER instrument, designed to analyze dust during special exploration missions. We detail the design of the DPU, which is based on a LEON3 processor and utilizes the GRLIB IP library, while exploring FPGA options for prototyping. We present the architecture and design synthesis results, along with the current status of project development.

The DPU will handle all software related to instrument management and status monitoring. This software must be robust and reliable to ensure mission success. Additionally, the DPU must manage the high data acquisition rates from the three instrument probes and process and transmit this data in real time.

We also emphasize the importance of using a model-based development flow manage the complexities of space projects. Selecting the right FPGA is crucial; factors such as cost, availability, radiation resistance, and compatibility with development tools must be considered. Utilizing the GRLIB IP library—comprising a comprehensive set of reusable IP cores designed for system-on-chip (SoC) development—offers advantages due to its vendor independence and compatibility with various CAD tools and target technologies. This flexibility facilitates integration into model-based development workflows.

ACKNOWLEDGMENT

Author Juan Manuel Gómez López acknowledges financial support from the Severo Ochoa grant CEX2021-001131-S funded by MCIN/AEI/ 10.13039/501100011033".

Funded by the European Union's HORIZON Research and Innovation programme under grant agreement No 101082466.

REFERENCES

[1] DUSTER (Dust Study, Transport, and Electrostatic Removal for Exploration Missions) https://duster.aeronomie.be/ HORIZON-CL4-2022-SPACE-01 - project 101082466.

[2] Schwan, J., Wang, X., Hsu, H. W., Grün, E., & Horányi, M. (2017). The charge state of electrostatically transported dust on regolith surfaces. Geophysical Research Letters, 44(7), 3059-3065.

[3] Necmi Cihan Orger, Kazuhiro Toyoda, Hirokazu Masui, Mengu Cho,Experimental investigation on silica dust lofting due to charging within micro-cavities and surface electric field in the vacuum chamber, Advances in Space Research, Volume 63, Issue 10, 2019, Pages 3270-3288, ISSN 0273-1177, https://doi.org/10.1016/j.asr.2019.01.045.

[4] Necmi Cihan Orger, Kazuhiro Toyoda, Hirokazu Masui, Mengu Cho, Experimental investigation on particle size and launch angle distribution of lofted dust particles by electrostatic forces, Advances in Space Research, Volume 68, Issue 3, 2021, Pages 1568-1581, ISSN 0273-1177, https://doi.org/10.1016/j.asr.2021.03.037

[5] Champlain, A., Matéo-Vélez, J. C., Roussel, J. F., Hess, S., Sarrailh, P., Murat, G., & Gajan, A. (2016). Lunar dust simulant charging and transport under UV irradiation in vacuum: Experimentsand numerical modeling. Journal of Geophysical Research: Space Physics, 121(1), 103-116

[6] Tyler M. Lovelly,∗ Travis W. Wise,† Shaun H. Holtzman,‡ and Alan D. Georg, Benchmarking Analysis of Space-Grade Central Processing Units and Field-Programmable Gate Arrays, JOURNAL OF AEROSPACE I NFORMATION S YSTEMS Vol. 15, No. 8, August 2018

[7] Ran Ginosar, SURVEY OF PROCESSORS FOR SPACE, Survey Space Processors DASIA2012

[8] F. Gaisler, “Leon3 Processor, “https://www.gaisler.com/index.php/products/processors/leon3”

[9] F. Gaisler, “NOEL-V Processor,” https://www.gaisler.com/index.php/products/processors/noel-v.

[10] M. W. Learn. Evaluation of the LEON3 Soft-Core Processor Within a Xilinx Radiation-Hardened Field Programmable Gate Array. Technical Report SAND2012-0454, Sandia National Labs, April 2011

[11] Gaisler. GRLIB VHDL IP Core Library, 2023

[12] Xilinx. Space Families https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-ultrascale-xqr.html

[13] Xilinx. Kintex UltraScale Family Overview https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-ultrascale.html

[14] OAR Corporation. RTEMS, An Open Real-Time Operating System. http://www.rtems.com, 2016.

[15] RTEMS pre-qualification toolkit https://rtems-qual.io.esa.int

WASM and Containers for Real-Time Serverless Edge Computing

Isser Kadusale, Gautam Gala, and Gerhard Fohler
Chair of Real-Time Systems,
Technical University of Kaiserslautern-Landau (RPTU)
isser.kadusale@rptu.de, gala@eit.uni-kl.de, gerhard.fohler@rptu.de

***Abstract*—**

A new trend of offloading real-time (RT) systems to edge computing platforms is emerging. Function-as-a-Service, a type of serverless computing, has useful applications for RT systems by offloading aperiodic tasks to the edge, or by reducing over-provisioned periodic tasks. Due to their comparatively low overheads, containers are an attractive alternative to VMs for supporting RT serverless computing. However, containers still have relatively long cold startup times. WebAssembly (WASM) has been suggested for use in non-RT serverless edge computing. Due to its short cold startup times, it can also be a potential alternative to containers. We explore this by running synthetic benchmark tests to see the timing performance of WASM runtimes in comparison to a container. We compare the startup times and execution times of different WASM runtimes and a container, with the goal of determining which of them is better in which situations. Furthermore, we discuss the qualitative suitability of WASM as compared to containers for RT serverless edge computing.

***Index Terms*—WebAssembly, Container, Real-time, Safety-critical, Cloud computing, edge computing, Serverless, FaaS**

I. Introduction

As seen in recent EU projects such as SECREDAS [1], a new trend of offloading real-time (RT) systems applications is emerging. Edge computing is becoming popular as it enables data processing on edge nodes that are located only a few network hops away. This minimizes and bounds the worst-case network latency, which is crucial for RT systems.

Previous works [1], [2], [3], suggested fine-tuned versions of cloud hypervisors (e.g., KVM [4]) for real-time cloud/edge nodes, allowing multiple operating systems/applications to run on the same physical machine in Virtual Machines (VMs). Hypervisors provide strong CPU-level temporal isolation and memory-level spatial isolation, in addition to security. However, VMs come with significantly higher resource usage and performance overhead.

Some works [5], [6], [7], [8] have explored containers (e.g., docker [9]) as a lightweight alternative to VMs, especially for supporting RT serverless/Function-as-a-Service (FaaS) computing. Serverless computing allows for automatic scaling of resources based on demand, saving costs and improving efficiency. FaaS allows developers to write and deploy code as individual functions to be executed in response to specific events, and can be used for offloading aperiodic real-time applications to the edge. Periodic RT applications can also use serverless computing to help reduce resource over-provisioning between periodic invocations.

Although containers have much lower runtime overheads (near-native application) compared to VMs, they can still suffer from large (100s of milliseconds) cold startup times (i.e., on first invocation) for initialization and resource provisioning [10]. A way to tackle cold startup problems for RT applications is by over-provisioning CPU and memory resources to keep container (or VM) instances warm between invocations [8], [1], [11]. However, this is contrary to the resource efficiency goal of serverless computing.

Previous work on non-RT serverless edge computing (e.g., [12], [13]) proposed using WebAssembly (WASM) [14], a binary instruction format, due to negligible cold startup times. WASM allows developers to compile code written in programming languages such as C++, Rust, and Go into a portable format that can be executed on multiple hardware and software platforms. WASM also has potential for use in RT systems as it runs code in a sandboxed environment, providing isolation and security like containers/VMs.

However, the WASM runtime environment can introduce additional overhead compared to native/container-compiled code. In addition, WASM is still a relatively new technology and needs more exploration to understand its suitability for RT systems.

Previous works [15], [16], [13], [14] have generically analyzed the performance of WASM runtime compared to containers for IoT or execution as native Linux application or execution of WASM code in the browser. We, on the other hand, use benchmarks to understand the difference between the cold startup times and runtime overheads for WASM runtimes (Wasmtime [17], WebAssembly Micro Runtime (WAMR) [18]) and docker containers. Our study is not about proving one technology's superiority over another. It's about understanding the suitability of Docker container or WASM for different use cases; and if WASM is better suited for a specific use case, we aim to identify which particular WASM runtime is the best for that use case. We performed an experimental evaluation on a server-grade edge node (Dell R640 [19]) with 2^{nd} Gen Intel Xeon Gold Processor [20] (2.3 GHz, 16 cores) and 8 GB RAM running Ubuntu 24.04.

The remainder of the paper is organized as follows: Sect. II presents related work on the suitability of WebAssembly for edge computing. Sect. III gives a brief overview of We-

bAssembly runtimes. Sect. IV describes our tests to compare the performance of WASM runtimes and docker, and presents the results. Sect. V discusses factors in deciding between WASM runtimes and containers. Sec. VI concludes the paper.

II. Related Work

Grosch et al. [21] position WebAssembly (WASM) as an essential enabling technology for designing distributed applications across the Edge-Cloud continuum. They proposed a theoretical edge-cloud framework based on WASM to support real-time applications and meet their safety, timeliness, and reliability requirements. Zaeske et al. [22] integrate a WASM interpreter in a safety-critical ARINC 653 Hypervisor and demonstrate the approach's feasibility by assessing the resultant binary size and performance.

Previous work, such as [23], [15], [13], [12], have proposed using WASM for serverless (edge) computing, esp. to support Function-as-a-Service (FaaS). Gackstatter et al. [13] argue that the high cold-start latencies of containers render them useless to support unpredictable and bursty workloads and may cancel the latency benefits that come with edge computing when serverless functions are deployed. Hall and Ramachandran [12] demonstrated how a WASM-based serverless platform may provide similar isolation and performance guarantees of container-based platforms while reducing average application cold start times and the resources needed to host them. Pham et al. [15], [24] evaluated WASM as an alternative to Docker containers for IoT applications. [15] demonstrated that WASM has a modest performance cost over Docker containers and provides security to applications. Kjorveziroski et al. [23] compared the cold start delays and total execution times of three WASM runtimes: WasmEdge, Wasmer, and Wasmtime to the performance of the containerd container runtime. They showed that WASM runtimes have better results in most tests, and Wasmtime is the fastest runtime among those evaluated. Jangda et al. [16] conduct a large-scale evaluation of the performance of WASM vs. native using the SPEC CPU suite of benchmarks. They showed that applications compiled to WebAssembly run significantly slower. Bosamiya et al. [25] implemented two techniques to produce provable safe WASM code. Their evaluation of these two techniques indicated that WASM can be leveraged to produce provably safe multilingual sandboxing with performance comparable to standard, unsafe approaches.

Contrarily, we use PolyBench/C [26] benchmark to understand better the difference between the cold start times and runtime overheads for WASM using three different runtimes and their different modes. We also compared cold start times and runtime overheads with docker containers. We focus on understanding workloads for which a particular WASM runtime excels or workloads where containers outperform WASM.

III. WebAssembly Runtimes

WebAssembly binaries are run by software called WebAssembly runtimes. There are many existing non-web environment WASM runtimes for executing WASM binaries. In this paper, we present the results of testing two WASM runtimes: Wasmtime, and WebAssembly Micro Runtime (WAMR). We also tested Wasmer, but, since it performed worse than the other two runtimes, we don't show its results to save space.

One difference between WASM runtimes and container runtimes (such as containerd), is that WASM runtimes are responsible for translating WASM binaries so that they can run on the target platform, whereas container images already have natively-compiled binaries.

WASM runtimes can also have different running modes: interpreter, Ahead-of-Time compilation (AOT), and Just-in-Time compilation (JIT).

When running in interpreter mode, the runtime executes the equivalent native instruction(s) for each WASM instruction. With AOT, the WASM binary is compiled into its native equivalent in advance, while with JIT, the compilation is done at runtime.

AOT offers the best performance in terms of startup and runtime execution, but this requires compiling the WASM binary for the intended platform, which trades off some flexibility of WASM. JIT retains this at the cost of some performance overhead, but not as much as interpreter mode, which is why we used JIT for our tests.

Runtimes can also have different compiler backends that affect their timing performance. Wasmtime uses Cranelift as its backend compiler. It aims for fast compilation speeds while generating code that performs almost as well as LLVM or gcc.

WAMR has two JIT tiers: Fast JIT and LLVM JIT. Fast JIT has a faster startup time compared to LLVM JIT, but at the cost of execution time performance. WAMR also supports runtime dynamic tier-up from Fast JIT to LLVM JIT, referred to as multi-tier JIT.

IV. Experimentation

In this section, we present our testing of different WebAssembly runtimes and Docker. The goal of these tests are to determine where WebAssembly is better than containers in terms of total execution time.

All tests were performed on a Dell R640 server with an Intel Xeon Gold 5218 processor and 8 GB of RAM running Ubuntu 24.04. The processor has 16 cores, 22 MB of cache, and has a max turbo frequency of 3.9 GHz but was limited to its base frequency of 2.3 GHz for our tests.

We used the PolyBench/C [26] benchmarks for our tests. It is a collection of benchmarks used in many papers evaluating WebAssembly performance.

We used the Alpine docker image as a base image for our test container. This image is based on Alpine Linux, a small and simple Linux distribution based on musl libc and busybox.

WebAssembly binaries were generated using WASI SDK (wasi-sdk-21), a WebAssembly C/C++ toolchain. Both WASM and native binaries were compiled with clang using the same optimization options (e.g., -O3).

A. WebAssembly runtimes and container comparison

We ran each PolyBench/C benchmark 25 times with the docker container and each runtime's available compiler.

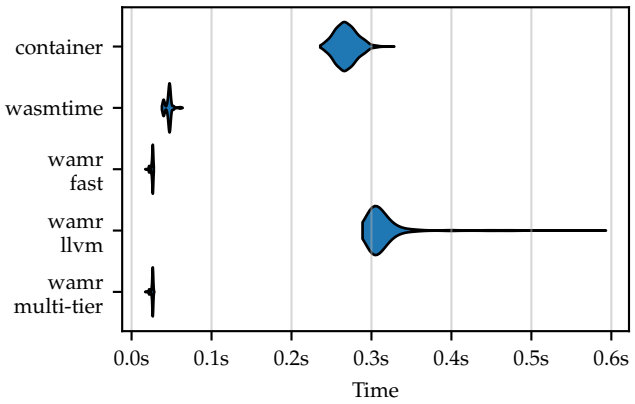


Fig. 1: Startup times.

When offloading RT tasks, the startup time must be included in the WCET, since it delays task completion. Figure 1 shows the startup time results. The startup times were relatively similar for each category regardless of the benchmark. WAMR's fast JIT and multi-tier JIT had the fastest startup times at around 26ms on average, followed by Wasmtime at roughly 46ms. WAMR's LLVM JIT's startup time averaged at around 315ms, which was slower than the container, whose average was around 268ms. It is, however, important to note that the startup time of a container can vary a lot depending on the configuration of the image [27]. Note also that this overhead is only present in the first invocation of a task, since WASM runtimes cache the compiled version of the binary, and containers don't need go through the same setup if they are kept live. Periodic tasks, in particular, only suffer this penalty at the start.

Figure 2 shows the total execution times (i.e., from starting the runtime/container until the end of `main()`) of the runtimes and the container for each benchmark. For benchmarks that last less than a second, WAMR's LLVM JIT and the container performed significantly worse compared to the others. This is largely due to their startup times, which were longer than the total execution time of the others. In most of the longer benchmarks, they outperform the other three.

B. Prolonged benchmark test

The previous test showed that the container and WAMR's LLVM JIT perform better in benchmarks that lasted longer, but this may be due to the workload of the benchmark itself, and not just the duration of the benchmark.

For this reason, we conducted a prolonged benchmark test where we ran the `main()` function of each benchmark multiple times continuously, and noted the time for each run. This simulates a periodic task over multiple jobs, or an aperiodic task with a long duration workload. Doing so allows us to see the trend of how the container and the runtimes perform over a longer time period. The first timestamp in this test is equivalent to the total duration that was measured in the previous test, while subsequent timestamps no longer have a startup period since the WASM runtime/container is not restarted.

To save space, we only show a few selected graphs in Figure 3 which show trends that are representative of what can be observed in other benchmarks. In 8 of the benchmarks, the container had the fastest times after a prolonged run, while in 20 of the benchmarks, either WAMR's LLVM JIT or multi-tier JIT had the fastest times.

V. Discussion

As shown in the previous section, WASM with fast JIT compilers have a clear advantage for short workloads because of its fast startup. This is not a big advantage for longer workloads, where having a more optimized binary is more important, as shown in our prolonged benchmark test. Thus, when deciding on what to use for offloading a task to an edge server, the expected duration of a task should be considered. WAMR's multi-tier approach is of particular interest since it attempts to get the best of both worlds by using a fast compiler at the start and switches to a slower compiler, which is better at optimizing, later on.

Contrary to our expectations, the container did not have the fastest times for the majority of the benchmarks in the prolonged test. This needs to be investigated further since it implies that WASM can perform better than natively-compiled code.

We also want to note that while synthetic benchmarks can be indicative of performance, they don't completely represent real-world workloads. Further tests that run workloads closer to real-world applications are needed.

A. Qualitative Analysis

a) Practical implications: There are other practical aspects to consider when deciding between using WebAssembly and containers.

Even a minimal container image like the Alpine docker image is 5 MB in size, while a WebAssembly binary could be as small as several kilobytes. This has implications on the bandwidth and storage usage of the intended application.

WebAssembly is also platform-independent. Having a single binary for different server platforms that an application may run on is a significant advantage. Edge servers may have different architectures, which necessitates different native binaries that will have their own overheads in terms of maintenance and certification.

b) Limitations of WASM runtimes: WebAssembly in non-web environments is still relatively early in development and does not yet have support for many features.

An example of this is setjmp/longjmp support, which is used in exception handling. This feature has not yet been standardized in WebAssembly, which makes it difficult or perhaps even impossible to use WebAssembly in the many applications that use it.

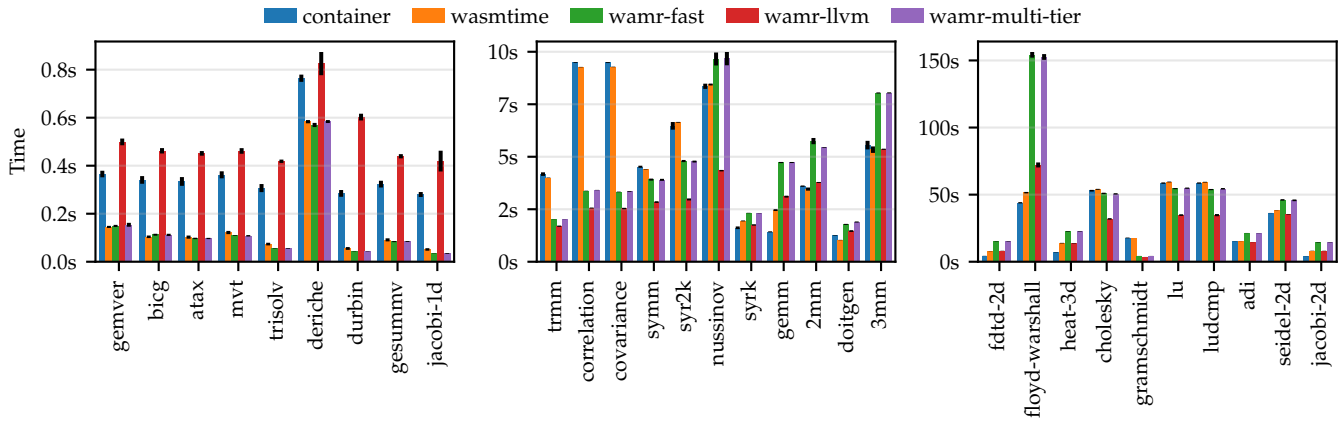
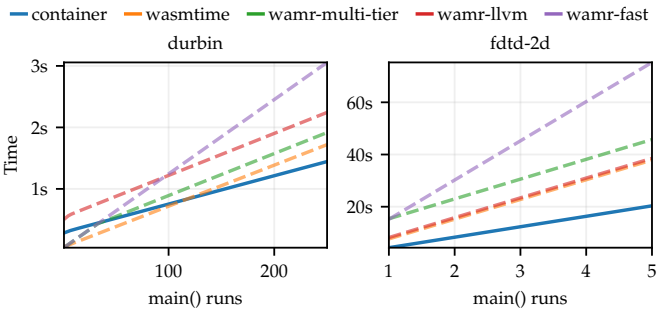
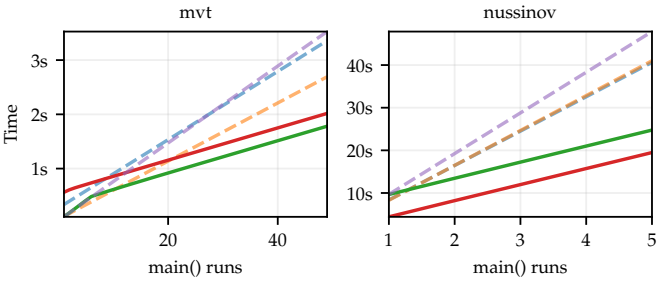


Fig. 2: Total benchmark times.



(a) Container is faster.



(b) WAMR LLVM JIT or multi-tier JIT is faster.

Fig. 3: Prolonged benchmark test results.

Another example is thread support. While the wasi-threads proposal has already been implemented in some WebAssembly runtimes, others have yet to add support for it. This limits the use of some WebAssembly runtimes for multithreaded applications.

These issues, including several others, were encountered when we attempted to compare WebAssembly and containers with a practical application, specifically the You Only Look Once (YOLO) framework used for real-time object detection [28]. We needed to work around these issues to get it to compile to WebAssembly. Due to these workarounds, we were not sure if it was still a valid test. For a sample object-detection test case, the native binary took about 300 milliseconds on average to make a prediction. In comparison, Wasmtime took about 2 seconds on average. WAMR took about 3.5 seconds on average, but it was only spawning 4 processing threads, as opposed to creating a thread for each core similar to Wasmtime and the native binary, even when specifying 12 as the max number of threads. Lastly, Wasmer crashes since it would reach the maximum number of open file descriptors because it would not close files that the application should only temporarily have opened.

As we noted earlier, these issues are likely due to WebAssembly still being relatively young, and most would likely be addressed in the future.

VI. Conclusion

In conclusion, our assessment of WebAssembly runtimes and containers for real-time serverless computing has highlighted the potential of WebAssembly for short periodic and aperiodic tasks at the edge. While shorter startup times make WebAssembly runtimes advantageous for specific applications, the significance of startup times diminishes for longer execution time tasks, where containers with optimized code compilers may offer better performance.

WAMR's multi-tier approach shows promise in addressing performance drop-offs for longer WCET tasks, presenting an interesting area for further research. A possible avenue for real-time serverless edge computing research is to see if it can combine WebAssembly's quick startup advantage with the better long-term performance of native binaries in containers.

When utilizing WebAssembly for real-time serverless computing, it's crucial to consider factors such as platform independence. This is particularly important as the technology continues to evolve and additional essential features are yet to be added.

Future work will focus on exploring why WAMR outperformed the container in prolonged benchmark tests. This investigation is crucial to understand whether this trend holds for different types of tasks. We also plan to extend our testing to real-world application workloads like the ECRTS ARM industrial challenge.

REFERENCES

[1] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, "RT-cloud: Virtualization technologies and cloud computing for railway use-case," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, 2021.

[2] L. Abeni and D. Faggioli, "An experimental analysis of the xen and kvm latencies," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 18–26.

[3] ——, "Using xen and kvm as real-time hypervisors," *Journal of Systems Architecture*, vol. 106, p. 101709, 2020.

[4] "Main page," 2023. [Online]. Available: https://www.linux-kvm.org/

[5] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[6] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[7] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Reducing temporal interference in private clouds through real-time containers," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 124–131.

[8] G. Monaco, G. Gala, and G. Fohler, "Shared resource orchestration extensions for kubernetes to support real-time cloud containers," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023, pp. 97–106.

[9] I. Docker, "Docker," *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[10] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[11] G. Fohler, G. Gala, D. Gracia Pérez, and C. Pagetti, "Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator," in *ERTS 2018*, ser. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan. 2018. [Online]. Available: https://hal.science/hal-01700860

[12] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[13] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[15] S. Pham, K. Oliveira, and C.-H. Lung, "Webassembly modules as alternative to docker containers in iot application development," in *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, 2023, pp. 519–524.

[16] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[17] B. Alliance, "Wasmtime: A fast and secure runtime for WebAssembly," last accessed:05/24. [Online]. Available: https://wasmtime.dev/

[18] ——, "WebAssembly micro runtime," last accessed:05/24. [Online]. Available: https://bytecodealliance.github.io/wamr.dev/

[19] May 2024. [Online]. Available: https://www.dell.com/us-en/work/shop/povw/poweredge-r640

[20] May 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html

[21] F.-J. Grosch, D. Dasari, N. Pereira, and A. Rowe, "Building reliable distributed edge-cloud applications with webassembly," in *Workshop on real-time cloud systems (RT-Cloud)*, 2022.

[22] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "Webassembly in avionics : Decoupling software from hardware," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023, pp. 1–10.

[23] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *J. Grid Comput.*, vol. 21, no. 3, jun 2023. [Online]. Available: https://doi.org/10.1007/s10723-023-09669-8

[24] J. Napieralla, "Considering webassembly containers for edge computing on hardware-constrained iot devices," 2020.

[25] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe multilingual software sandboxing using WebAssembly," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1975–1992. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[26] L.-N. Pouchet and T. Yuki, "Polybench/C 4.2.1," *URL: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1*, 2024.

[27] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023, pp. 94–105.

[28] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," 2022. [Online]. Available: https://arxiv.org/abs/2207.02696

How to prove the existence of an optimal frequency assignment algorithm for CPU and memory accesses in absence of an appropriate energy model

Myriam Mabrouki*, Liliana Cucu-Grosjean*, Stéphan Plassart†
*Kopernic, Inria, France, Email: firstname.lastname@inria.fr
† University of Savoie Mont-Blanc, France, Email: firstname.lastname@univ-smb.fr

***Abstract*—While targeting a reduced energy consumption for the embedded real-time systems community, different techniques exist and, in this paper, we are interested in Dynamic Voltage and Frequency Scaling. Quan and Hu propose an algorithm ensuring an energy-optimal CPU frequency assignment for executing independent programs on a single-core processor under real-time constraints. Since its optimality is proposed for CPU variable frequencies, we propose to add variable frequency assignment for memory accesses to the problem solved by Quan and Hu. Based on numerical evaluation of energy consumption of TACLeBench programs, we study the impact of both CPU and memory accesses frequencies on the energy consumption but no existing energy model considers these two factors. Moving back to the existence of an optimal algorithm to assign both frequencies, one may propose to find the appropriate pair of CPU-memory accesses frequencies, but an appropriate energy model is required to compare two different pairs.**

I. Motivation, our problem and existing results

Since reducing energy consumption is a major concern for the embedded real-time systems community, techniques like Dynamic Voltage and Frequency Scaling (DVFS) [1] and Dynamic Power Management (DPM) [2] have been developed. In particular, DVFS consists of reducing processor voltage and frequency to reduce power consumption while meeting timing constraints. Nonetheless, in the literature this technique mainly considers the processor frequency (or the CPU frequency) and not the memory frequency. In this paper, we study the impact of memory accesses on energy consumption and discuss the existence of an optimal frequency assignment algorithm for CPU and memory accesses.

Our problem More precisely, we consider a *real-time system* composed of a set τ of n independent, synchronous, implicit deadline, periodic tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$ scheduled by a preemptive fixed-priority scheduling algorithm on a single core processor.

Each task τ_i is defined by its worst-case execution time (WCET) C_i, and its period T_i equal to its deadline. An execution or instance of a task τ_i is called job. The execution time of the j^{th} instance, i.e. the j^{th} job, of task τ_i is denoted c_j. The execution time of a job is not necessarily equal to the WCET of the task. Since we deal with periodic tasks, a task releases a job at any instant $k * T_i$, where $k \in \mathbb{N}$.

Within the paper, unless stated otherwise, we consider a *fixed-priority algorithm*. Without any loss of generality, the priority assignment is given: the tasks are ordered in the decreased order of their priorities, i.e., τ_i has a higher priority than τ_j if $i < j$. Tasks are executed on a processor with one core and one memory storage, each of which has an associated execution frequency, i.e., new operations are performed at each new cycle. The processor frequency is denoted as f_p and the memory frequency is denoted as f_m. Moreover, f_p and f_m vary by having discrete values respectively within the interval $[0, f_p^{max}]$ and $[0, f_m^{max}]$ with f_p^{max} maximum processor frequency and f_m^{max} maximum memory frequency. Since an execution time of a task τ_i depends on CPU and memory frequencies, we denote it as $c_i^{f_p, f_m}$.

In this paper, we consider that the hardware features of the processor are such that we cannot modify the frequency during the execution of a program to match realistic hardware components. Thus, within our model of real-time system we consider that both frequencies, memory and processor are fixed during the schedule.

Existing work To the best of our knowledge, there is no result providing a solution of finding an optimal frequency assignment for both CPU and memory accesses to minimize energy consumption for the real-time systems considered in this paper. If one considers only the CPU frequency, then Hong et al. [3] propose a heuristic scheduler, while Quan and Hu provide a heuristic [4] and then a speed schedule for a fixed-priority real-time system, based on the offline algorithm of Yao et al. [1], that leads to a minimal energy consumption [5]. Moreover, these results consider that the CPU frequency is modifiable during the execution of tasks. Interestingly, Kim et al. [6] show that the lowest processor and memory frequencies do not always imply the lowest energy consumption.

Organization of the paper: We provide a detailed description of our experimental environment, then we propose a discussion on obtained energy consumption measures as well as the identification of future work.

II. Description of experimental environment

We consider a real-time system composed of tasks or programs of the benchmark TACLeBench [7] executed on a Raspberry Pi 3 model B+ [8]. This micro-controller has a 64-bit ARM Cortex-A53 quad-core processor, whose frequency should vary from 600 MHz to 1400 MHz. The memory is an *SDRAM*. The default minimum and maximum values set for the memory frequency are respectively 400 MHz and

500 MHz [9]. The operating system used is *Raspberry Pi OS*, a Linux operating system based on *Debian*.

In order to monitor the energy consumption, we use the power analyser *Otii Ace Pro* [10] and its associated software, *Otii 3 Desktop App* [11].

There are three steps in our experiments. The *first one* consists of creating several test programs with different quantities of memory accesses. The *second step* involves building a real-time system to study the impact of CPU and memory access frequency on the energy-consumption and the *third step* consists of measuring the energy consumption of our test programs and our system.

Building test programs We build four test programs. The first two programs are test programs that are used to control the impact of memory accesses, and the last two are actual applications on a realistic benchmark, TACLeBench [7].

The first program, called $test\_mem$, involves allocating a two dimensional array and doing write accesses on each cell. These accesses are made in a way to maximise cache misses and, thus, memory accesses. Indeed, if at the iteration k we access the cell at the position (i, j), with i representing the row number and j the column number, then we access the cell at the position $(i+1, j)$ at the iteration $k+1$. The second program, called $test\_cpu$, consists of looping as many times as there are cells in the array of the previous test. In this way, we have two control tests against which to compare energy consumption results.

The last two consist of executing one application of the benchmark TACLeBench [7] in a loop, however they differ with their sensitivity to the CPU frequency. The third program is *statemate*, executed in a loop of 100 iterations. It is chosen among all the TACLeBench tasks, because its execution time is less sensitive to the CPU frequency. Thus, this program is performing more memory requests (see Section III.B in [12]). The fourth program is *ammunition*. For this task, the execution time is more sensitive to the CPU frequency and should have less memory accesses [12]. Contrary to *statemate*, it is executed in a loop of 10 iterations due to its larger execution time.

All these tests are set to the highest priority with the `set_priority` function. Moreover, we transmit a message to the power analyser through the Universal asynchronous receiver transmitter (UART) protocol [13] at the beginning and at the end of each test. These messages are used to get precise timestamps of the beginning and the end of each measure.

Description of the real-time system Last section consider specific programs in isolation, however our real-time system is more complex, with several tasks. Therefore, we build a program called *global_system* composed of three tasks of TACLeBench [7] described in Table I : *statemate*, *ndes*, and *cjpeg_wrbmp*. These tasks are chosen since their execution times are less sensitive to the processors speed and, therefore, their number of memory accesses is higher than other tasks (see Section III.B in [12]).

The experimentation is done in two steps: First, the execution time of these three tasks are measured at the maximum

TABLE I: Task parameters of the systems

Task	i	$c_i^{f_p=1400, f_m=500}$	T_i
statemate	1	1.60 ms	9.86 ms
ndes	2	1.61 ms	9.86 ms
cjpeg_wrbmp	3	1.72 ms	9.86 ms

CPU and memory frequencies. Every application is executed on a single core with the `sched_setaffinity` function. To ensure our measures are as reliable as possible each program is set to the highest priority in the user-space with the `set_priority` function. Moreover, we disable, via *raspi-config* and *config.txt* the desktop mode, the camera module, Bluetooth, Wi-Fi and interfaces options (Serial Peripheral Interface (SPI) [14], Inter-Integrated Circuit (I2C) [15], Wire [16], Virtual network computing VNC [17], and Remote General Purpose Input/Output (GPIO)). Each application is executed 500 times. The execution time is measured with the `perf` command. Averages, standard deviations and 99% confidence intervals are then computed [18]. The execution times are the upper limit of the confidence intervals.

Second, we implement a scheduler [18]. The scheduler executes 1000 times a loop corresponding to the hyper-period of the real-time system. At the end of a loop, we check whether the tasks have been completed on time. If a deadline is missed, we end the program, thus the result is not considered in our experimentation. At the beginning and the end of the scheduler, we transmit a message to the power analyser with the UART protocol as in the test programs. Applying Quan and Hu's speed schedule to our system gives us a single interval with an associated speed which is the initial speed, thus the initial frequency, divided by two.

Energy consumption measures We power supply the power analyser *Otii Ace Pro* [10] with an adjustable power adapter. We connect the power analyser to *5V* and *Ground* pins of the micro-controller Raspberry Pi 3B+ through banana to jumper wires and to a computer through a USB wire. We also connect *RX* and *Digital Ground* pins of the power analyser respectively to *TX* and *Ground* pins of the micro-controller to capture UART logs. We use the software *Otii 3 Desktop App* [11] to switch on the power analyser which switches on the Raspberry Pi 3B+.

Since the micro-controller has a quad-core processor, we disable three of them to get a single core by setting `maxcpus=1` in */boot/firmware/cmdline.txt*.

In order to see how energy consumption evolves and to figure out a pattern, we measure the energy consumption under different frequency assignments for CPU and memory. We call a *configuration* a pair of CPU frequency and memory frequency (f_p, f_m). For each configuration, we measure our programs 100 times.

The configurations (f_p, f_m) we choose for our test programs *test_cpu, test_mem, statemate* and *ammunition* are the ones with f_p varying from 200 MHz to 1400 MHz in steps of 200 and f_m varying from 200 MHz to 500 MHz in steps of 100. The ranges chosen for the memory frequency are similar to the ones used in [12].

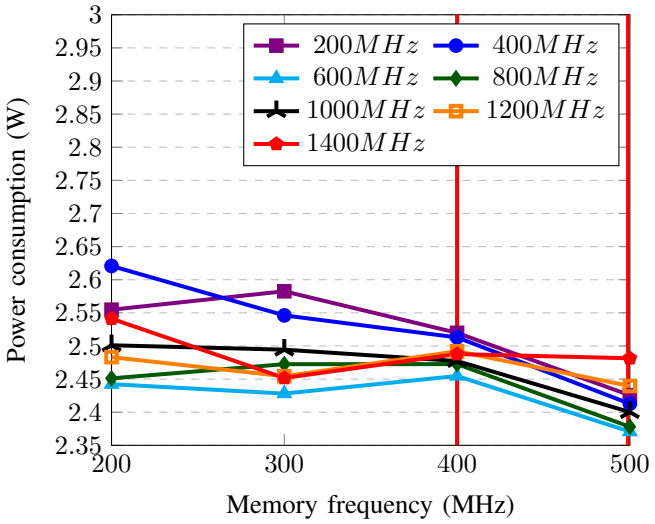


Fig. 1: Power consumption according to memory frequency at different fixed CPU frequency for *test_cpu*

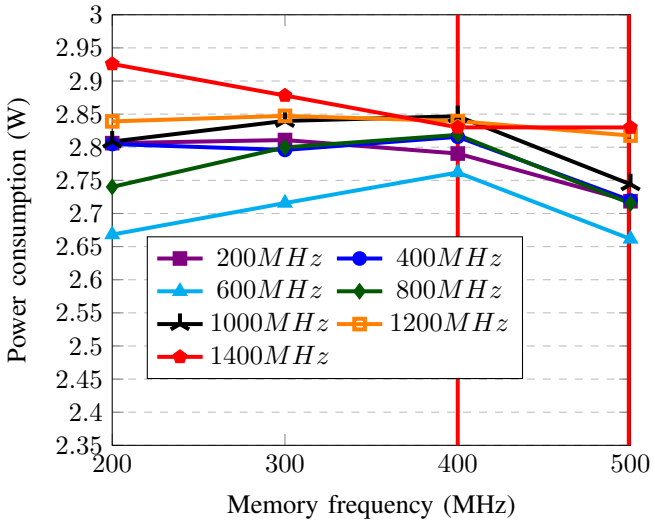


Fig. 2: Power consumption according to memory frequency at different fixed CPU frequency for *test_mem*

For the *global_system* program, the CPU frequency is set up to 700 MHz according to Quan and Hu's algorithm output. We measure the energy consumption under different memory frequency assignment; $f_m \in \{200, 300, 400, 500, 600\}$.

CPU frequency is set with `arm_freq` in */boot/firmware/config.txt*. Since we do not want the frequency to lower during the execution, we set `arm_freq_min` in */boot/firmware/config.txt* with the same frequency. In the same way, memory frequency is set with `sdram_freq` and `sdram_freq_min` in */boot/firmware/config.txt*.

Although, lowering the CPU frequency under 600 MHz does not results in power savings, we keep configurations where the CPU frequency is below 600 MHz to observe power consumption patterns, especially when the memory frequency is equal to or higher than the CPU frequency. The same goes for the memory frequency, even if it is not explicitly mentioned that memory frequencies below 400 MHz are not supported. In addition, we go above the indicated maximum frequency of the memory for the *global_system* program since we can overclock the SDRAM in order to have an additional power consumption value.

III. Discussion on energy consumption results and future work

The results obtained are in two CSV files [18] for reproducibility purpose. One is about power consumption results and the other contains the UART log. A power consumption is given every 20 microseconds. We compute the average power consumption of each measures and then we compute the average power consumption between all measures [18]. To do that, we consider power consumptions whose timestamps are included between the timestamps "end" and "begin" given in the UART log file. Timestamps in the UART log file are given in milliseconds. It is not the same precision as the timestamps in the power consumption file but is sufficient for our measures

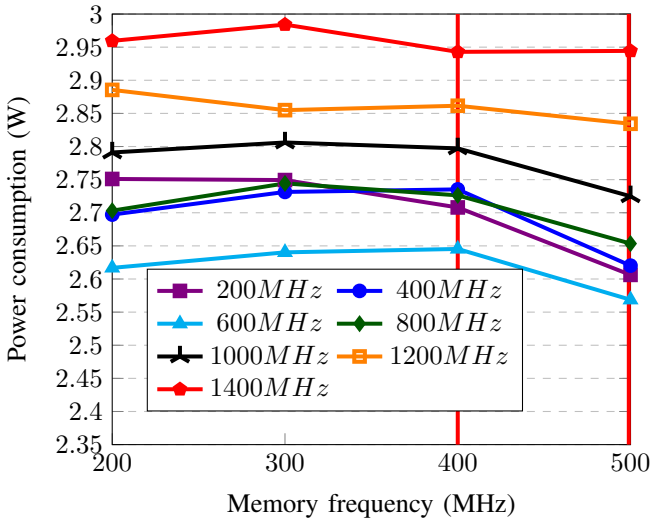


Fig. 3: Power consumption according to memory frequency at different fixed CPU frequency for *ammunition*

since the power consumption does not vary significantly during one millisecond. In addition, standard deviations, minimum and maximum are also computed [18].

When we look at the results of our test programs (Figures 1–4), we first see that the program *test_mem* is more power consuming than *test_cpu* as expected since *test_mem* does the same as *test_cpu* but with more memory accesses. Moreover, *ammunition* is more power consuming than *statemate*. The program *ammunition* has less memory accesses so the variations of the power consumption is mainly due to CPU frequencies. Conversely, with *statemate*, the memory frequency seems to have more impact than the CPU frequency on the power consumption. It is consistent with the characteristics of this task which has more memory requests.

The Rasberry Pi3 model B+ has a supported memory

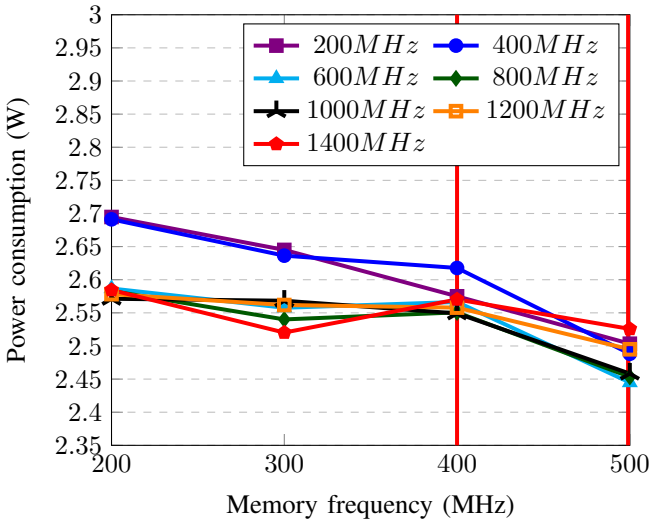


Fig. 4: Power consumption according to memory frequency at different fixed CPU frequency for *statemate*

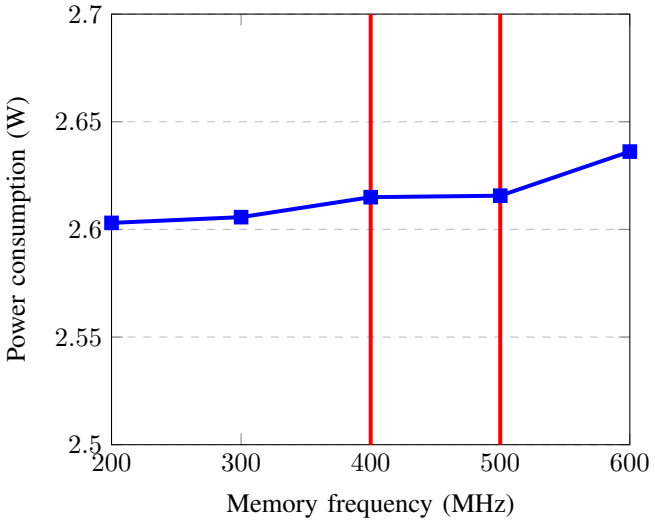


Fig. 5: Power consumption according to memory frequency at a CPU frequency of 700 MHz for *global_system*

frequency range of 400MHz to 500MHz [9] (these limits are represented by the vertical red lines in Figures 1–5). On this specific interval, one can observe that no matter the CPU frequency, the power consumption obtained at higher memory frequency, i.e., f_m = 500 MHz, is equivalent or lower than the power consumption obtained at lower memory frequency, i.e., f_m = 400 MHz.

When we consider lower CPU frequencies, the lowest power consumption is almost every time associated with the highest memory frequency. We observe that this is never the lowest memory frequency, nor the one closest to the CPU frequency that leads to the minimal energy consumption. Most of the time, the patterns of the power consumption in Figures 1–4 are either decreasing curves, or curves that increase and then decrease. Based on these results, there is a single memory frequency (this frequency differs according to the programs and the CPU frequency) that leads to the highest power consumption which means that, starting from this frequency, we can either increase or decrease the memory frequency to lower the power consumption. Given that lowering the memory frequency increases the execution time [12], it is better to increase the memory frequency to lower the total energy consumption. However, sometimes the curves of power consumption decrease, increase and then decrease again. Moreover, the power consumption decreases with the CPU frequency until the minimum default value which is f_p = 600 MHz as expected since lowering the CPU does not results in power savings [9].

As for the power consumption of the considered real-time system, we observe that the behaviour of the power consumption differs from the one of the test programs. Indeed, we can see in Figure 5 the power consumption is increasing with the memory frequency. If we link this result to the analysis done in the previous paragraph, it could mean we should increase the memory frequency to get the memory frequency leading to the highest power consumption. Notwithstanding, the power consumption is increasing very slowly. The lowest power consumption in this case is obtained when the memory frequency is set to 200 MHz and equals 2.60 W. The highest power consumption is obtained when the memory frequency is set to 600 MHz and its value is 2.64 W. Although both power consumptions are very close, we can get a lower power consumption. Therefore, there is a gain of 1.5% in power consumption when exploiting the impact of memory frequency.

Yet, in terms of gain in power consumption the highest we get are with *statemate* at f_p = 400 MHz with a reduce rate of 7.4% or with *test_cpu* at f_p = 400 MHz with a reduce rate of 8%. Indeed, with *statemate* we get a power consumption of 2.69 W at f_m = 200 MHz and 2.49 W at f_m = 500 MHz. For *test_cpu*, the power consumption equals 2.62 W at f_m = 200 MHz and then decreases to 2.41 W at f_m = 500 MHz. However, *test_cpu* is not supposed to have a lot of memory accesses.

Nevertheless, the measures show huge variations of power consumption, in spite of the pattern of the power consumption for each program. Indeed, the power consumption varies from 1.8 W to 4 W. Even when the CPU is idle, the power consumption varies around the same values.

Our discussion indicates that an energy model considering both CPU and memory accesses frequency is not straightforward. Moving back to the existence of an optimal algorithm to assign both frequencies, one may propose an exhaustive search algorithm by considering all possible pairs of CPU-memory accesses frequencies in order to identify the one with the lowest energy consumption. Nevertheless, an appropriate energy model is required to compare two different pairs within this exhaustive search algorithm. In conclusion, we identify as future work the proposition of an appropriate energy model as mandatory condition towards the proposition of an optimal algorithm.

REFERENCES

[1] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 374–382. IEEE Computer Society, 1995.

[2] Yann-Hang Lee, Krishna P. Reddy, and C. Mani Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS 2003), 2-4 July 2003, Porto, Portugal, Proceedings*, pages 105–112. IEEE Computer Society, 2003.

[3] Inki Hong, Gang Qu, M. Potkonjak, and M.B. Srivastavas. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 178–187, 1998.

[4] Gang Quan and Xiaobo Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 828–833. ACM, 2001.

[5] Gang Quan and Xiaobo Hu. Minimum energy fixed-priority scheduling for variable voltage processor. In *DATE*, pages 782–787. IEEE Computer Society, 2002.

[6] Young-Jin Kim and Jihong Kim. Exploration of memory-aware dynamic voltage scheduling for soft real-time applications. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), 17-19 August 2005, Hong Kong, China*, pages 177–180. IEEE Computer Society, 2005.

[7] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[8] Raspberry pi 3 model b+. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-3-model-b, 2024.

[9] config.txt. https://www.raspberrypi.com/documentation/computers/config_txt.html, 2024.

[10] Otii ace pro. https://docs.qoitech.com/user-manual/otii/hardware/otii-ace-pro#datasheet, 2024.

[11] Otii 3 desktop app. https://docs.qoitech.com/user-manual/otii/software, 2024.

[12] Roberto Medina and Liliana Cucu-Grosjean. Work-in-progress: Probabilistic system-wide DVFS for real-time embedded systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 508–511. IEEE, 2019.

[13] He Chun-zhi, Xia Yin-shui, and Wang Lun-yao. A universal asynchronous receiver transmitter design. In *2011 International Conference on Electronics, Communications and Control (ICECC)*, pages 691–694, 2011.

[14] Cliff Wootton. *Serial Peripheral Interface (SPI)*, pages 335–349. Apress, Berkeley, CA, 2016.

[15] Jayant Mankar, Chaitali Darode, Komal Trivedi, Madhura Kanoje, and Prachi Shahare. Review of i2c protocol. *International Journal of Research in Advent Technology*, 2(1), 2014.

[16] Sashavalli Maniyar. 1-wire® communication with pic® microcontroller. In *Application note AN1199*. Microchip Technology Inc, 2008.

[17] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[18] https://github.com/Energy-Memory/Energy-Memory-Acesses, 2024.

Host-Based Allocators for Device Memory

1st Oren Bell
Washington University in St Louis
St Louis, USA
oren.bell@wustl.edu

2nd Ashwin Kumar
Washington University in St Louis
St Louis, USA
ashwinkumar@wustl.edu

3rd Chris Gill
Washington University in St Louis
St Louis, USA
cdgill@wustl.edu

***Abstract*—Segregated Fit and Two-Level Segregated Fit are two common dynamic memory allocation algorithms used in real-time systems, due to their constant-time allocation and deallocation of memory.**

We pose a model where these allocation algorithms run on a host system but manage device memory. This model is exemplified in accelerated applications without unified memory that copy data in and out of device memory before and after a kernel call. Managing device memory from the host device incurs the following constraint: the allocator can't read the memory it is managing. This means we are unable to use boundary tags, which is a concept that has been ubiquitous in nearly every allocation algorithm, including segregated fit and two-level segregated fit. In this paper, we propose alternate designs to work around this constraint, and discuss in general the implications of this system model.

***Index Terms*—GPU, FPGA, hardware acceleration, heterogeneous computing, memory management**

I. Introduction

This paper concerns itself with real-time dynamic memory management, i.e., algorithms to manage a heap of memory that clients can request and free blocks from at any time, and do so in O(1) time. The field of dynamic memory management could be said to have been started by Knuth [1]. Some 30 years of progress is well summarized by Wilson and Johnstone [2][3] in their comprehensive survey. Since then, further improvements have been made [4][5][6][7], but overall the subject can be considered to be highly mature.

Most existing memory management algorithms include metadata as headers or footers within allocated blocks. In contrast, our system model assumes that the allocator is running on a host system, managing device memory. This may occur, for example, if part of a computation is offloaded to a GPU or FPGA device while the rest of the computation runs on a multicore processor. This in turn implies that the compute device cannot (conveniently and efficiently) access the memory it manages. Therefore, any data needed by the allocation algorithm cannot be stored in its allocated blocks.

We motivate this system model by considering the usecases surrounding how device memory is managed. In the current state of the art, memory allocation/deallocation is either done on the peripheral compute device in often proprietary device drivers. This causes memory allocators to be treated as a fixed black box.

However, different applications may have differing needs to manage their memory. The most optimal allocator may be high performant, real-time constrained, or have domain-specific characteristics catering to the application. Fixing the allocator choice in proprietary drivers and hardware denies developers the choice to optimize this aspect of their program through selective mapping of portions of the application and its supporting libraries to different computational devices. Examples of such applications include drone cinematography [8] and real-time hybrid simulation experiments in earthquake engineering [9].

For our work, we assume that existing hardware and drivers are used to allocate arbitrarily large blocks of memory, but finer-grained memory allocation is then done in userspace by the host machine. As was mentioned previously, these allocators are constrained by the inability to read the memory they are managing and cannot store metadata in allocated blocks. We present alternative algorithms that overcome this constraint.

In Section II, we propose alternative measures to overcome this constraint of being unable to read managed memory. In Section III, we present our updated alternative allocation algorithms. We compare the performance of one of our algorithms to the default CUDA memory allocation functions in Section IV. Finally, we conclude in Section V with thoughts on implementations and usecases for this work.

II. Alternatives to Free Lists

Our paper explores a model in which device memory is managed from the host. This prevents the allocator from reading the memory being managed, which creates a new challenge not present in traditional memory allocation schemes. We cannot use boundary tags, a standard approach that was first mentioned by Knuth [1].

The traditional usage of boundary tags is illustrated in Figure 1. A free list is formed by a linked list of blocks that are available for reuse. The header points to the next element in the free list, and the footer points to the header of the same block, which enables coalescence between two adjacent blocks in O(1) time. After a block is freed, one can subtract the footer size from the block's address to obtain the address of the header of the prior block. From there, one can check if that prior block is in the free list, and if so, the two will be coalesced. The next block in the free list can also be checked to see if it's adjacent in memory, possibly coalescing a total of three blocks.

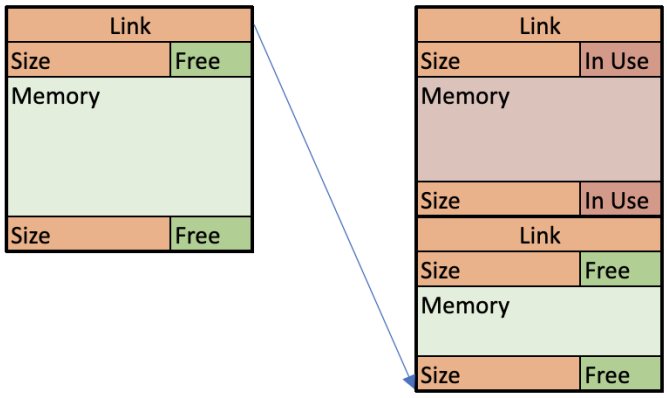


Fig. 1: Demonstration of Boundary Tags

A. Hash Tables

The strategy we propose is to use a hashtable, keyed by device memory address, to store all the information typically found in the header and footer of an allocated block.

The traditional algorithms for free-list traversal and coalescence operation would need to be modified to accommodate an extra step to lookup data in the hashtable. Additionally, the in-use blocks would also need to be tracked in the hashtable as well, as they also have necessary information in their header and footer.

With the use of hash tables, we aim to achieve O(1) allocation and deallocation, so co-mingling free and in-use blocks will not be a major performance concern.

The key for the hash table is the address of a block. The value contains i) the block size, ii) the address of the next block in a free list, iii) the address of a previous block in a free list, iv) the address of the previous block adjacent in address space. Thus we can create, in effect, free lists that span across a hash table. The ability to easily maintain multiple free lists is useful for segregated fit algorithms.

The primary downside of this approach is the overhead. Each entry contains 6 words (the key, block size, two references for a doubly linked list, one reference for a prior adjacent block, and a reference for a separate list linking collisions in the hash table). This means that in the worst case scenario, minimum allocation size of one word, the overhead is $\sim 83\%$. This is comparable to the worst case overhead of 80% seen in doubly linked free lists in conventional host-only algorithms. However, we only recommend this approach for applications with a fewer small allocations, such as those with a larger minimum block size, or any applications for which segregated fit algorithms are applicable.

B. Bitmasks

One way to minimize overhead is through the use of bitmasks. In this approach, device memory is divided equal-sized chunks, and the only overhead is a single bit in host memory to indicate whether that block is free or not. If we consider an allocation size lower bound of 8 bytes to be the worst case scenario, this creates an overhead of only 1.5% (one bit for 8 bytes). That produces 16MB of overhead for a gigabyte of device memory.

Allocation is done by finding a contiguous string of set bits in the bitmask corresponding to the desired size. For example, to allocate 1kB of memory, it is necessary to find a string of 128 bits that are all ones, incurring an O(n) cost for allocation, on the order of the address space size. The `clz` and `ffs` hardware instructions can be used to accelerate a bitsearch such as this.

All these bits must then be set to zero to indicate their corresponding blocks are in use. When the block is freed, they are all returned to ones (coalescence is implicit here). This means allocation and deallocation operations also incur an O(s) cost, on the order of the block size.

Bitmasks have low overhead and relatively high time complexity on the order of allocation size. Because of the low overhead, a potential use-case is to use bitmasks to manage a pool of small memory chunks, either of the same size (as in an object pool) or commingled varying sizes, as described above.

In order to achieve the real-time O(1) allocation and deallocation, the size of the bitmask must be bounded. For instance, a `clz` and `ffs` can search a 64-bit string in a single operation. This feature is also used in segregated fit algorithms that maintain multiple free-lists of different size classes. The bitmask serves as an availability mask to indicate which free-lists are non-empty.

III. Segregation Algorithms

A. Segregated Fit

In segregated fit, multiple free lists are maintained in size buckets for different powers of 2. Given a requested size to allocate, it is rounded up to the nearest order of magnitude, and then the first block from that list is selected. This means (assuming no empty free lists) the selected block may be nearly 4x larger than the requested size, causing 75% fragmentation, as discussed in prior literature [2] [3].

All lists are initialized pointing at a null block: an invalid block in the hashtable is meant to indicate the end of a free list. Accompanying this array of lists is an availability bitmap that can indicate which lists are non-empty. If the free list for a desired size class is empty, the next eligible list can be found by using the find-first-set (ffs) bit operation on the bitmap.

Our implementation stores the free lists in a hashtable, as discussed in Section II-A. Algorithm 1 and 2 show pseudocode for allocation and deallocation, respectively.

B. Two-Level Segregated Fit for Device Memory

Two-Level Segregated Fit[10][11][6] (TLSF) is a real-time dynamic memory allocation algorithm. Its primary benefits are reduced fragmentation and O(1) allocation and deallocation. It is an extension of segregated fit, dividing class sizes not only logarithmically, but also linearly, in a two-tiered system.

The allocation and deallocation steps are functionally the same, except the lookup step requires consulting 2 bitmasks.

Algorithm 1: Allocate memory in segregated fit

```
Data: free_lists, bitmask, hashtable, size ≥ 0
▷ Find available free list large enough to accommodate
requested_bins ← 1 ≪ ceil(log2(size));
order ← ffs(requested_bins);
                ▷ Lookup head of candidate list in hashtable
it = ht[free_lists[order]];
free_lists[order] ← it.next_free;
it.free = False;
                          ▷ Split off surplus portion of block
if size < it.size then
    new_block.size ← it.size − size;
    new_block.free ← True;
    new_block.addr ← it.addr + size;
    new_block.prev_adj ← it.addr;
    new_block.prev ← 0;
    it.size ← size;
       ▷ Place new block at head of free list in its size
     class bin_idx ← floor(log2(new_block.size));
    new_block.next = free_lists[bin_idx];
    new_block.prev = 0;
    ht[new_block.next].prev = new_block.addr;
    free_lists[bin_idx] = new_block.addr;
    bitmask ← bitmask | (1 ≪ bin_idx);
    ht[new_block.addr+new_block.size].prev_adj ←
     new_block.addr;
                                  ▷ Insert into the hashtable
    ht[new_block.addr] = new_block;
ht[it.addr] = it;
return it.addr;
```

Algorithm 2: Deallocate memory in segregated fit

```
Data: free_lists, bitmask, hashtable, addr ≥ 0
it ← ht[addr];
left ← ht[it.prev_adj];
right ← ht[it.addr + it.size];
if left.free then
    if right.free then
                          ▷ Coalesce all 3, erase it and right
        left.size+ = it.size + right.size;
        ht[right.addr + right.size].prev_adj ←
         left.addr;
                                          ▷ Update bitmask
        if right.prev = right.next then
            unset_bit ← 2^floor(log2(right.size));
            bitmask ← bitmask & ¬unset_bit;
        remove(right);
    else
                                  ▷ Coalesce left block, erase it
        left.size+ = it.size;
        right.prev_adj ← left.addr;
    if it.prev_free = it.next_free then
        unset_bit ← 2^ceil(log2(it.size));
        bitmask ← bitmask & ¬unset_bit;
    remove(it);
    it = left;
else if right.free then
                                          ▷ Coalesce right block
    it.size+ = right.size;
    ht[right.addr + right.size].prev_adj ← it.addr;
                                          ▷ Update bitmask
    if right.prev = right.next then
        unset_bit ← 2^floor(log2(right.size));
        bitmask ← bitmask & ¬unset_bit;
    remove(right);
       ▷ Put coalesced block in free list in its size class
bin_idx ← floor(log2(it.size));
it.next = free_lists[bin_idx];
ht[it.next].prev = it.addr;
it.prev = 0;
free_lists[bin_idx] = it.addr;
it.free = True;
ht[it.addr] = it;
bitmask ← bitmask | 2^bin_idx;
```

One is logarithmic and functions just as detailed in Algorithm 1 and Algorithm 2. TLSF differs in that this points at another bitmask which subdivides the space linearly, as illustrated in Figure 2. This tier then points to a free list of blocks in that size class. If the free list is non-empty the linear bitmask will have a corresponding 1 set. If the linear bitmask is non-zero, then the logarithmic bitmask will have a corresponding 1 set.

Free list manipulations are done the same way as in segregated fit, so Algorithms 1 and 2 only need to be modified to account for this two-tiered lookup process and bitmask manipulation.

C. Hybrid Allocators and Object Buffers

Hybrid approaches may also be employed, using object buffers to manage object pools for allocations smaller than a page (<4kB), and segregated lists for larger allocations.

Object pools allow for the efficient management of vast amounts of small allocations, but are not a practical approach for larger allocations. A hybrid approach can permit the benefits of both approaches.

It is important to consider the practical use of such a hybrid approach in this context. A host-based allocator for device memory is unlikely to allocate large numbers of small objects, but rather large memory blocks that are passed to a kernel call for a hardware-accelerated device. If small allocations are created, they would be intermediary memory allocated by device code, likely using a SIMD allocator such as XMalloc[5] or ScatterAlloc[7]. These allocators are designed to handle many allocation requests in parallel without the need for locks, a usecase that doesn't apply to the host-based model we have presented in our paper.

So, our usecase would be best served by limiting the smallest granularity to a medium sized value, such as 4kB,

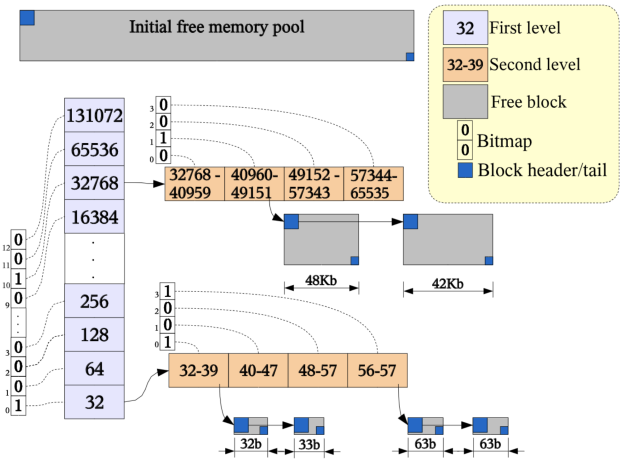


Fig. 2: TLSF Data Structure[11]

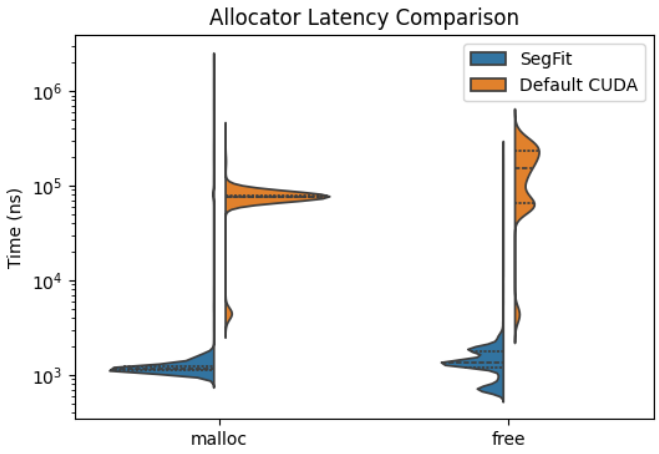


Fig. 3: Latency of malloc and free

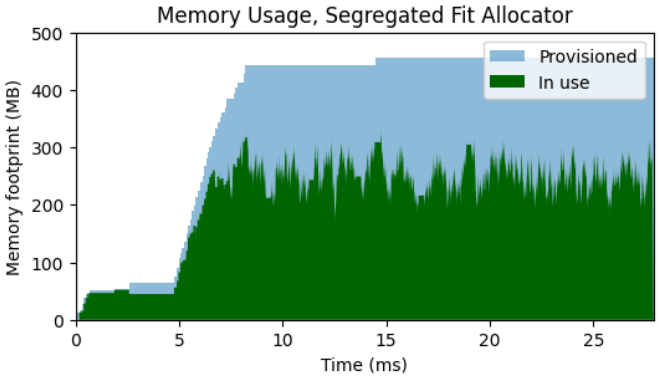


Fig. 4: Memory usage during benchmark

and exclusively using a segregated fit algorithm.

IV. Evaluation

To evaluate, we wrote an implementation of Algorithms 1 and 2. We make use of CUDA driver calls to perform initial allocation of physical memory when the memory pool doesn't have an adequate candidate block. After mapping the memory into virtual address space, it is treated as a contiguous and fungible memory pool. When a process terminates, all physical blocks are unmapped.

We compare our work to the default CUDA allocator. We adapted the malloc-large benchmark from the MiMalloc benchmark suite[12] to use an implementation of our segregated fit algorithm. This algorithm randomly allocates and frees blocks 1kB and 16MB in size, several thousand times.

As a baseline, we also ran the benchmark with ordinary calls to cudaMalloc and cudaFree. The order and size of the malloc and free operations was consistent between our work and the baseline. Latency comparisons between the two are shown in Figure 3.

Our work outperforms the default CUDA allocator by 2 orders of magnitude. There are some outliers during program initialization where the segregated fit malloc can take upwards of 2ms, but these disappear as the program settles into a steady state.

We measure fragmentation in our experiments (Figure 4) by tracking not only the memory in use by the benchmark, but also the physical device memory provisioned to the benchmark process. In our segregated fit allocator, we observe a roughly 50% fragmentation ratio. This is consistent with prior work[2] on segregated fit. A Two-Level Segregated Fit allocator would see improved fragmentation[6].

V. Conclusion

We note that prior work on dynamic memory allocation relies on free lists and boundary tags. We've posed a heterogeneous system model where memory is stored on one device and managed from another. After observing that existing algorithms are no longer applicable, we've provided alternatives to free lists and updated versions of classic sequential and segregated fit algorithms.

An implementation of our segregated fit algorithm was tested against conventional calls to cudaMalloc and cudaFree. It was found that, after a warmup period, our work consistently outperformed the CUDA calls by two orders of magnitude. This is mainly due to its implicit and efficient memory recycling.

We do not claim to outperform established GPU allocators [13] at scale, nor to address their highly parallelized use-cases. However, in specific circumstances where kernels are not expected to allocate memory and diverse accelerated computation is combined into a large heterogeneous application, performance benefits can be achieved by managing memory on the host. It is unnecessary to invoke device calls to free memory, and instead it can be dynamically recycled.

Acknowledgements

This work was sponsored by NSF Grant 2229290

REFERENCES

[1] D. E. Knuth, "The art of computer programming, vol 1: Fundamental," *Algorithms. Reading, MA: Addison-Wesley*, 1968.

[2] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[3] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *International Workshop on Memory Management*. Springer, 1995, pp. 1–116.

[4] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger, "A compacting {Real-Time} memory management system," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[5] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 2010, pp. 1134–1139.

[6] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: A new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* IEEE, 2004, pp. 79–88.

[7] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "Scatteralloc: Massively parallel dynamic memory allocation for the gpu," in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–10.

[8] R. Bonatti, W. Wang, C. Ho, A. Ahuja, M. Gschwindt, E. Camci, E. Kayacan, S. Choudhury, and S. Scherer, "Autonomous aerial cinematography in unstructured environments with learned artistic decision-making," *Journal of Field Robotics*, vol. 37, no. 4, pp. 606–641, 2020.

[9] J. Condori, A. Maghareh, J. Orr, H.-W. Li, H. Montoya, S. Dyke, C. Gill, and A. Prakash, "Exploiting parallel computing to control uncertain nonlinear systems in real-time," *Experimental Techniques*, vol. 44, pp. 735–749, 2020.

[10] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, no. 2, pp. 149–179, 2008.

[11] M. Masmano, I. Ripoll, and A. Crespo, "Dynamic storage allocation for real-time embedded systems," *Proc. of Real-Time System Simposium WIP*, 2003.

[12] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 2019, pp. 244–265.

[13] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on gpus slow? a survey and benchmarks," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 219–233.