Dynamic Mitigation of Multi-Core Interference in Safety-Critical Systems

Moritz Meier [1], Wanja Zaeske [1], and Umut Durak [1]

Abstract: Over the last two decades, multicore architectures have become the de facto standard for processors. With the avionics industry lagging roughly 10 to 20 years behind, it is only since the early 2020's that multicore architectures are pursued more seriously, as evidenced for example by the issue of AMC/AC 20-193 by the EASA/FAA in 2022 and 2024 respectively. While they enable many advantages such as increased compute density and higher bandwidth between processes, they also come with the big disadvantage of degraded isolation between cores which can cause interference in the temporal behavior of programs. For safety-critical applications relying on predictable timing that is a major problem. This paper explores a novel approach to mitigate multicore interference, relying on continuous measurement of computational progress using WebAssembly fuel, enabling dynamic mitigation of the multicore interference effects — irrespective of the specific interference channel.

Keywords: multicore, interference, safety-critical, real-time, avionics, WebAssembly

1 Multicore Interference

Modern, Commercially Off-The-Shelf (COTS), computing platforms are dominated by multicore processors or system-on-chips, packing many processing units like Central Processing Unit (CPU)s, Graphics Processing Unit (GPU)s, accelerators, etc. into a single chip to elevate compute density and bandwidth between processes while achieving better efficiency in terms of power, size and weight in comparison to single-core processors. In contrast, safety-critical systems like avionics still mostly rely on single-core architectures and are only slowly starting to utilize COTS multicore architectures, motivated by the aforementioned. However, one reason for slow adoption is the problem that real-time properties are much harder to guarantee in multicore systems.

While the execution units in a multicore architecture work mostly independent of each other, they do rely on many shared resources like memory controllers, caches and bus interconnects. These shared resources are on one hand the foundation for multicore architectures and their performance and efficiency gains but on the other hand also a source of temporal non-determinism, due to scheduling and allocation policies implemented in hardware which are needed for those shared resources. Common interference channels are for example shared caches where one core can evict data used by another core and therefore cause cache misses, resulting in spikes in the access latency, often orders of magnitude worse

[1] German Aerospace Center (DLR), Institute of Flight Systems, Lilienthalplatz 7, 38108 Braunschweig, Germany, moritz.meier@dlr.de, https://orcid.org/0009-0004-0241-8125;
wanja.zaeske@dlr.de, https://orcid.org/0000-0002-1427-2627;
umut.durak@dlr.de, https://orcid.org/0000-0002-2928-1710

than on cache hits. In safety critical systems where real-time properties are required, such uncertainties in timing are often unacceptable.

Resource	Mechanism	Effect
Shared Cache	Eviction of still-used data	Increase in Latency for Memory accesses (Data Accesses and Instruction Fetches -> CPU pipeline stall)
Cache Coherency Protocol	Temporally unavailability of cache lines	Increase in Latency for Memory accesses
Interconnect	Congestion & Unfair arbitration	Increase in Latency, especially also for IO operations, possibly starvation of CPU cores
Memory Bus	Bandwidth saturation	Increase in Latency for cache refills and write-backs
Memory Controller	Row buffer contention	Increase in Latency for cache refills and write-backs

Tab. 1: Common interference channels in COTS hardware

Due to the inherent architecture of multicore processors it is not possible to fully prevent interference without impinging on the multicore advantages. A survey paper from Lugo et al. [Lu22] gathers different approaches and techniques for reducing multicore interference. It labels over 200 referenced papers based on a taxonomy, illustrated in Figure 3 of [Lu22]. In this taxonomy, techniques aimed at reducing interference are grouped by three distinct causes of interference: main memory, cache and (memory-) bus related interference. This demarcates an important gap in most techniques (i.e. as described in [Lu22]): mitigation only of (a) specific interference channel(s).

Not being agnostic makes the aforementioned techniques less suitable for COTS hardware in which a constellation of arbitration and allocation policies over shared resources (and thus potential interference channels) is present. Often these are only poorly (or not at all) documented. Furthermore, many techniques (such as static cache partitioning) lead to performance degradation [2]; particularly in cases where parallel execution does not cause interference at all or where the interference in a channel remains within acceptable bounds. In consequence, diminishing returns from the application of multicore processors question their application to begin with.

While the mitigation techniques summarized by Lugo et al. [Lu22] are mostly static or design-time based, the question arises whether dynamic mechanisms could work and provide a viable alternative. Further on, Zaeske et al. hint at a possibly exploitable connection between WebAssembly (Wasm) fuel and multi-core interference [Za25], motivating a second

[2] The SP-IMPact paper from Costa et al. [Co25] shows how, for example, cache coloring (a cache partitioning technique) causes performance degradation in general — due to less cache being available — and how cache partitioning can help to mitigate multicore interference. Results from the paper show that there is an optimum between no partitioning, therefore interference is not mitigated and performance is only impacted by interference, and full partitioning, where all interference is mitigated but the loss of cache memory causes a severe performance degradation.

research question: can Wasm fuel be used to implement dynamic multi-core interference mitigation?

2 Mitigating Multi-Core Interference at Runtime

The main challenge in interference mitigation is to reliably perform a set amount of computation in a given time, despite the presence of other processes running in parallel on other cores. Instead of mitigating individual interference channels, another approach is the mitigation of only the end-to-end observable effects in timing caused by interference, on the application of interest.

The core-software and/or execution environment like an operating system, a hypervisor etc. may continuously observe the progress of a primary, *real-time critical task* — a task being a contiguous, fixed amount of computation and real-time critical referring to the existence of a timing deadline that has to be met — and estimate whether the desired deadline will be met. If said runtime determines the task to meet its deadline, then no significant interference impacts the critical task, thus no mitigation is necessary.

On the other hand if the runtime determines a likely deadline miss, i.e. due to interference causing slower than anticipated execution progress, then the runtime can actively intervene by pausing secondary, non-critical, interfering tasks, eliminating the interference sources. The secondary tasks then either stay suspended until the primary task finishes or are resumed again once the runtime determines that the primary task will achieve its deadline with sufficient margin.

This observation and mitigation would happen at a level below the usual process scheduling (e.g. ARINC 653), and only considers tasks that are actually running in parallel on different cores. The only goal of this mitigation strategy is to ensure a task can (at least to a statistically degree) meet its deadline. This guarantee could then be used to implement a classical process scheduling, based on Worst-Case Execution-Time (WCET) analysis, on top.

The advantage of such dynamic mitigation is that the mitigation targets not individual interference channels themselves (like with most techniques presented in [Lu22]), but only the potential effects of those interference channels on a process. The approach thus implicitly covers all known and unknown interference channels at once [3], decoupling the technique from hardware architecture implementation details, enabling the use of COTS hardware.

The disadvantage in comparison to unmitigated multicore systems is that at any given time only one real-time critical tasks can run, all other parallel running tasks have to be considered not real-time critical, otherwise the critical tasks could again cause interference in each other. Hence, the number of executable critical tasks remains identical to a single active core system, however the remaining cores become usable for non-critical tasks.

[3] assuming however, that the interference channel is directly caused by software

This approach could be formalized in terms of a control loop, where the controlled variable is the execution progress of the primary task with the interference being modeled as a disturbance. The actuated variables are the execution states of the secondary tasks which are paused and resumed, resulting in a bang-bang controller. More sophisticated control schemes, i.e. using performance and efficiency cores or fine-grained down-clocking, are certainly thinkable but escape the scope of this paper.

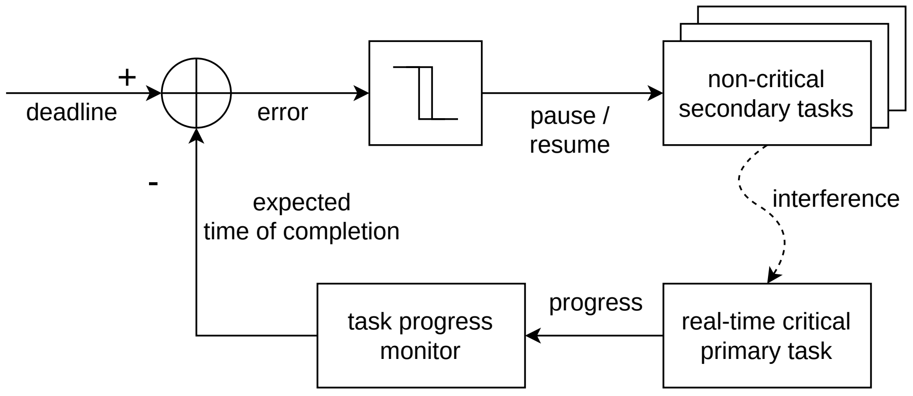
+
deadline
error
-
pause / resume
non-critical secondary tasks
interference
expected time of completion
progress
task progress monitor
real-time critical primary task

Fig. 1: Control loop for dynamic interference mitigation

3 Task Progress Monitoring

As stated previously, our approach's critical mechanism is the runtime being able to observe the computational progress of the primary, real-time critical task, in order to estimate whether the task can satisfy its deadline. This requires the runtime for one to be aware of the worst-case execution length of the critical task (e.g. how much computational work is required to complete the task) and the deadline in terms of WCET. Additionally, a fine-grained runtime mechanism to observe the computational progress of the critical task is needed too. The worst-case execution length of a task could be derived at compile time, at least for a restricted subset of programs that prohibits recursions and unbounded loops[4], by analyzing the longest execution trace to determine the total amount of fuel necessary to complete the program. The WCET has to be specified at design-time.

The progress observation of a task at runtime can be implemented in multiple different ways:

A cooperative approach would rely on the task itself to report its progress, for example via compiler generated checkpoints. This is a very intrusive mechanism, as the task itself has to be instrumented, possibly leading to complication e.g. with certification (and qualification

[4] which should be an acceptable limitation in safety-critical tasks

of the tools), and requires the OS to trust the task regarding its self reported progress [5]. Nonetheless this approach is already patented: Zlatanchev et al. described a mechanism where tasks are composed of micro tasks which emit trace data upon completion to observe computational progress [ZH18].

The alternatives are preemptive mechanisms where the runtime slices the execution of a task into chunks with a known execution length and measures the execution-time. This eliminates all the disadvantages of the cooperative approach but requires the real-time critical task to be interrupted in regular intervals. This would of course decrease the performance of the task but the added delays due to the measurement should be of constant size and can be relatively small compared to the size of the execution slices, so that the influence is a constant already known at design-time.

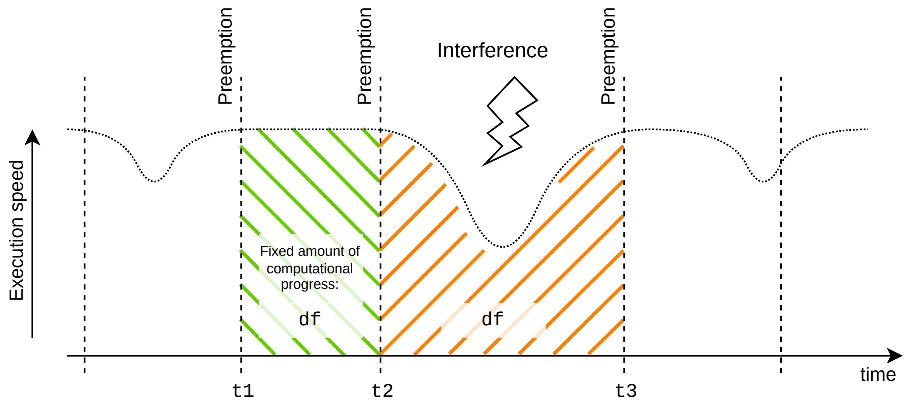

Fig. 2: Execution of a critical task in discrete slices, interrupted by the runtime to observe the computational progress. Y-Axis shows the execution speed; each unit area under the curve represents a fixed amount of computational work. Execution slice A between t_1 and t_2 is not affected by interference, execution slice B between t_2 and t_3 is. Both execution slices represent the same amount of computational progress df, but slice A is executed faster, then slice B ($t_2 - t_1 < t_3 - t_2$).

This paper employs only the preemptive approach, since this approach does not require the modification of executables and is therefore much better suited for safety-critical applications. We identified two potential mechanism that are suitable to implement such a preemptive progress observation and estimation:

One is the Performance Monitoring Unit (PMU) which exists as hardware module in most modern processor implementations and provides a variety of different metrics such as how many instructions where executed, how many CPU cycles where used, how many memory accesses and cache hits and misses occurred, and might[6] provide the ability to generate a hardware interrupt after a fixed number of instructions were executed. While

[5] otherwise the task may starve other tasks by never reporting any computational progress
[6] Some PMU specifications and implementations do, some (e.g. Cortex-A53) do not.

these capabilities are certainly interesting for observing the progress of a task, it also comes with the disadvantage that the PMU is by nature very architecture and even implementation specific, resulting in tight coupling to the hardware.

A more generic approach might be the use of an interpreter. Instead of running native tasks directly on the hardware, the runtime would include an interpreter which executes a program in byte-code form and monitors the progress by measuring the throughput of byte-code instructions per unit time. The obvious disadvantage here is the performance penalty of an interpreter, but an interpreter can also provide a multitude of other advantages, especially to safety-critical applications such as stricter and more formalized isolation, benefits in testing and certification, cross-platform support, etc. according to Zaeske et al. [Za23].

One particularly interesting interpreter technology is Wasm [22]. Because of its original use-case in client-side web applications it offers very strict isolation properties, validation at runtime, both of which are desirable features for high assurance systems like avionics [Za23].

Another desired feature of Wasm is its *fuel* system. Although currently not part of the Wasm specification [22], most common Wasm interpreter implement a fuel system to limit the amount of compute a (potential hostile) web application can use. With this fuel system the host process specifies a certain amount of fuel to the interpreter before it is invoked. With each byte-code instruction executed, some fuel is consumed, in the simplest implementation each byte-code instruction requires exactly one fuel unit. Once the fuel is used up the interpreter returns control flow into the host process, enabling it to either cancel the execution of the byte-code or to resume the execution with additional fuel. In summary, Wasm fuel enables both measurement of computational progress, but also scheduling [7].

Hence it fits very well into the preemptive progress observation approach. A runtime can schedule a real-time critical task in a Wasm interpreter and use the fuel system to slice the execution into chunks with known execution length (e.g. fuel). In between each slice the interpreter returns into the runtime which can then look at the computational progress, i.e. the amount of fuel consumed and the corresponding time elapsed in order to estimate the time-of-arrival of the task. If the estimated time-of-arrival is larger than the aspired deadline, the runtime can take action by pausing the non-critical tasks, eliminating the root causes of interference (see Figure 1).

4 Linux Demonstrator

To assess this approach a proof-of-concept demonstrator was developed and evaluated. To simplify reproduction, this demonstrator is based on a mainstream x86-64 workstation running a mainline Linux kernel. While this may not be sufficient to prove that dynamic

[7] i.e. it is trivial to implement a token bucket traffic shaper/scheduler on top of it

mitigation of multicore interference in real-time critical systems is possible and viable, it suffices to refute any conceptual errors.

The demonstrator consists of a central control process which spawns a single primary child-process, simulating a real-time critical task, as well as multiple secondary intruder child-processes — simulating non-critical tasks — to cause multicore interference in the primary process. The primary and all secondary processes use core pinning in order to fix the execution and thereby the interference channel constellation in place and core isolation is used to significantly reduce the noise originating from the Linux kernel on those cores.

The primary process runs a Wasm interpreter with the already introduced preemptive, fuel-based mechanism for progress observation and estimation. The Wasm interpreter which is used in this demonstrator is developed by the Institute of Flight Systems at the German Aerospace Center (DLR) in Braunschweig [Ge26b] and is especially targeted towards use-cases in avionics [Za25].

This Wasm interpreter currently implements a very simple fuel system, where each executed byte-code instruction consumes exactly one fuel unit.

The secondary cores run a StressNG [Ki25] instance to stress the caches, which without mitigation in place causes significant interference in the primary process.

In order to implement a full mitigation loop, a mechanism is needed, so that unacceptable slow-down of the primary process can pause and resume the secondary processes (which cause severe interference). This is implemented in the central control process. The primary process requests throttling of interference source from the control process through shared-memory. These requests are then realized by the parent control processes through the use of Linux Control-Groups (CGroups), which allow freezing and thawing of the secondary processes all at once, with low latency.

5 Evaluation

For the evaluation the demonstrator uses the fuel-based progress observation to determine how much time dt is used to run an execution slice of length df (e.g. fuel amount). From there a metric of time-per-fuel $\frac{dt}{df}$ with unit $[\mathrm{ns}/\mathrm{fuel}]$ can be established. The evaluation only considers this metric instead of absolute measurements like execution-time and execution length to make the evaluation less dependent on the task, the selected slice length (fuel amount) and various other parameters.

For the mitigation a target time-per-fuel value is set, that the mitigator tries to achieve. For a real use-case this would be derived from the defined worst-case execution-time and the worst-case execution length (e.g. worst-case execution fuel) which can be determined at compile-time.

To evaluate the concept multiple questions should be answered:

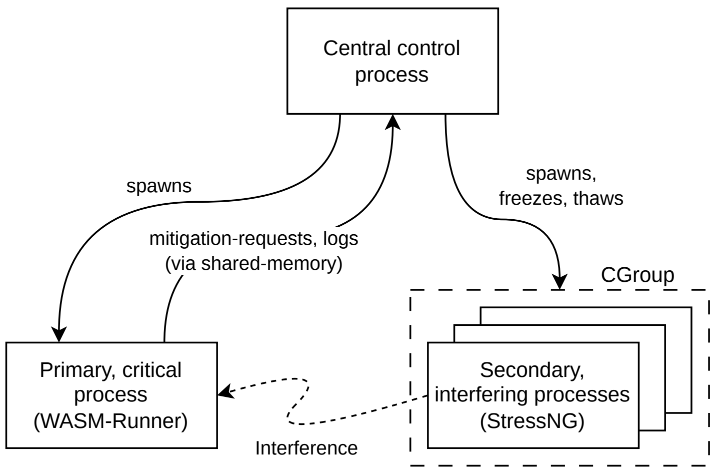
Central control process
spawns
mitigation-requests, logs (via shared-memory)
spawns, freezes, thaws
CGroup
Primary, critical process (WASM-Runner)
Secondary, interfering processes (StressNG)
Interference

Fig. 3: Demonstrator implementation. A central process spawns primary and secondary processes. The primary process measure the interference and issues mitigation requests to the control process. This control request then acts on the requests by freezing and thawing the secondary, interference causing processes via CGroups.

1. Can the preemptive, fuel-based progress observation be used to detect the effects of multicore interference?
2. Can the effects of multicore interference be effectively mitigated by utilizing a control loop, modulating the interference sources?
3. How efficient is the mitigation relatively to an unmitigated multicore system?
4. How much overhead does the fuel-based progress observation cause?

To answer the first question 1 one can compare the distribution of time-per-fuel measurements on singular execution slices in Table 2 in an unmitigated (e.g. with full interference) and a fully-mitigated (all interference sources disabled) case. While the bulk of the samples have similar distribution, the vastly different total execution-time hints that most interference has to be manifested in the outliers, i.e. some execution slices are not affected at all, while others are heavily affected by the interference. This makes sense in the context that critical and non-critical tasks run without any synchronization, therefore interference events can be considered pseudo random and execution slices vary naturally in utilized memory and cache bandwidth. Comparing the distribution of only the affected slices shows that the median time-per-fuel for samples that are affected by interference is roughly twice as high as in the fully, statically mitigated case (all interference sources disabled).

This suggests that the fuel-based metering approach can be used to detect interference, but it might be preferential to average over multiple or all slices (either by applying a moving

Description	w/o Interference	with Interference
Mean Time-per-Fuel, total	19.8 ns/fuel	29.3 ns/fuel
Median Time-per-Fuel, total	9.8 ns/fuel	9.9 ns/fuel
Median Time-per-Fuel, only affected slices	310 ns/fuel	610 ns/fuel

Tab. 2: Empirical measurement of the interference

average or by averaging over all previous slices), in order to capture a full picture, because of the interference effects not being distributed evenly across measurements.

The next question 2 is whether dynamic mitigation at runtime is possible? To evaluate that multiple ($N = 100$) runs of the primary, Wasm based task were executed — in parallel with the interference-causing intruder tasks — with three different setups of mitigation. The baselines are again the unmitigated and fully mitigated (all interference source disabled) settings, and a new setting for dynamic mitigation at runtime is introduced.

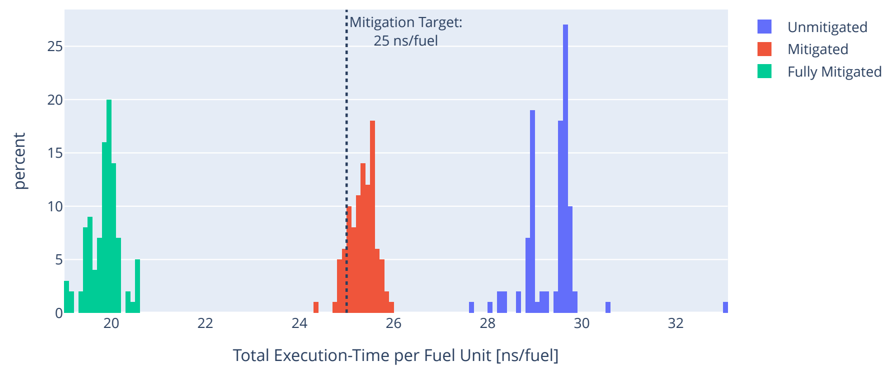

Fig. 4: Dynamic mitigation results, $N = 100$ runs each, for the mitigation a WCET of 25 ns/fuel was targeted

Figure 4 shows that in unmitigated case it takes about 29 ns/fuel to execute a single fuel unit and in the fully mitigated case it takes about 19 ns/fuel. This is consistent with the result from the previous test. The dynamically mitigated setup results in 25.5 ns/fuel. Given the pre-defined target set-point of 25 ns/fuel in the setup it shows that the mitigation technique works as a concept. With additional tuning of the margins in the control loop the remaining error of 0.5 ns/fuel between the targeted and measured mean time-per-fuel could be removed. A realistic use-case would require an application dependent trade-off to be made between control margins and retained multicore advantages.

Another interesting visualization is the averaged execution-time per fuel versus time in an exemplary recording of a single run.

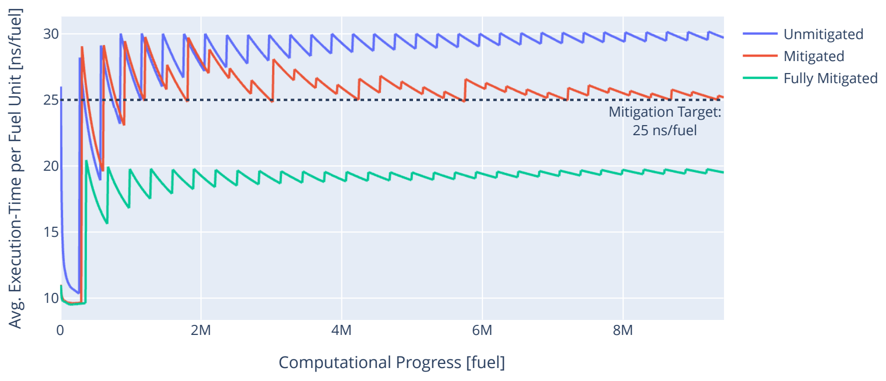

Fig. 5: Single test run, showing the trend of the avg. execution-time per fuel over computational progress

Figure 5 shows how the avg. execution-time per fuel converges over time. Especially it shows how the dynamically mitigated setup starts out similar to the unmitigated setup but over time converges against the targeted set-point of 25 ns/fuel. The reoccurring spikes are probably caused by the Linux host environment, other possible sources like the measurement and interpreter implementation were considered and ruled out.

Additionally the mitigator activity can be plotted, to visualize when mitigation is active, e.g. the interference sources are frozen. This can be used to evaluate how efficient the mitigation is. One goal of this dynamic mitigation strategy is to preserve multicore gains at least partially, otherwise this approach would be pointless if the secondary cores are not running.

Figure 6 shows that in order to achieve the mitigation target the secondary, interference causing tasks must be frozen most of the time. Only less than 10 % of the execution-time in the secondary task is preserved. This is very likely due to StressNG [Ki25] being used as interference source, which is designed to cause interference by deliberately trashing the caches, therefore presenting a worst-case adversary. This answers question 3 regarding the efficiency of an mitigated vs unmitigated system.

The figure 7 above shoes the efficiency of the fuel-based observation approach (Question 4). The efficiency here relates to how much time is spent in the interpreter running the task itself versus how much time is spent in total, running the task and the interpreter. For execution slices larger than 10 k fuel units, peak efficiency is reached. Smaller slices than that degrade the efficiency, because context switches into the runtime and back occur more frequently. On a modern x86-64 machine a slice of 10 k fuel units is executed in roughly 100 µs, there is probably not much use for making these slices even smaller.

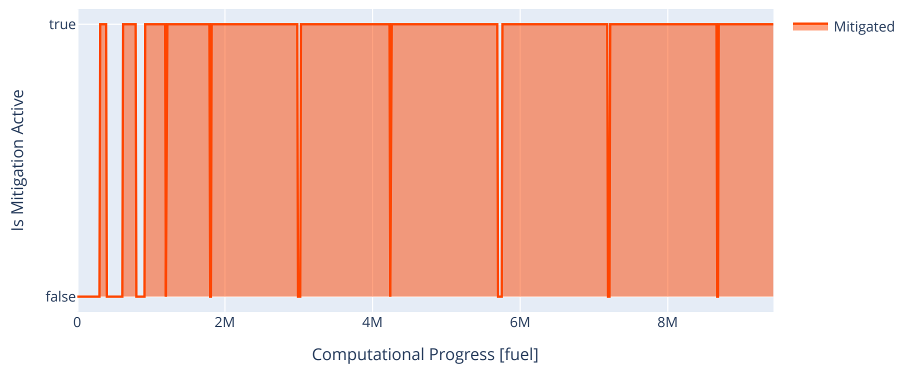

Fig. 6: Mitigation activity over computational progress, mitigation being active corresponds to secondary tasks being frozen

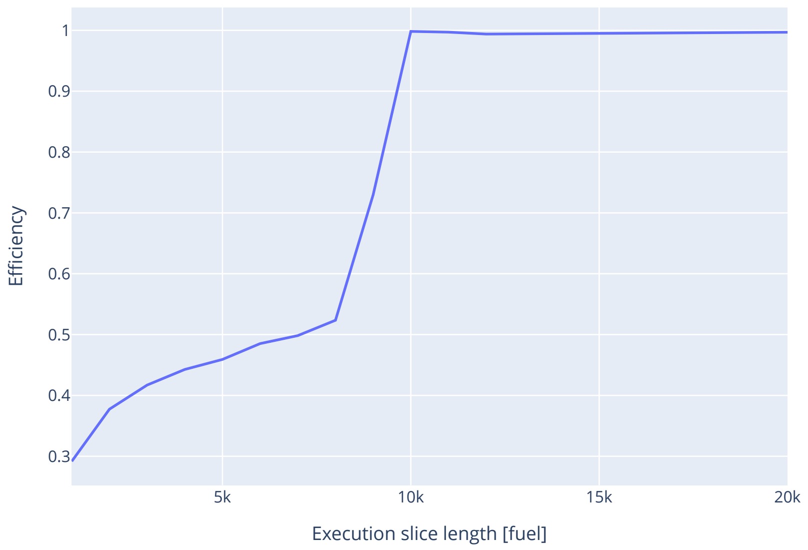

Fig. 7: Efficiency of the fuel-based metering in the interpreter

6 Known Limitations

While this paper covers an area of ongoing research, certain limitations already become evident. This section briefly summarizes them as starting points for further investigation.

- **Interpretation is slow**. In simple benchmarks we observed an order of magnitude increase in run-time when comparing execution in an ad-hoc interpreter vs. execution with Ahead of Time (AOT) compilation of WebAssembly to native object code (for a detailed elaboration of ad-hoc vs. AOT compilation for interpreters refer to [Za25, ch. 1]). This finding is consistent with other performance evaluations, i.e. the one performed by Titzer et al. in [Ti22]. As fuel support is also prsent in AOT compiling interpreters like Wasmtime, we however suspect this performance overhead to be avoidable, at the cost of re-tuning the interference mitigation control loop.
- **Not all Wasm instructions are $O(1)$**. Earlier, we assume that each Wasm instruction is associated with a given amount of fuel. This does not hold, for example the `memory.copy` instruction (similar to `memcpy` in C) takes an operand which controls the number of bytes to be copied. Naturally, the computational cost to execute the instruction can not accurately be described by a fixed fuel cost.
- **The mitigation technique does not enable parallel execution of critical applications**. This forms a fundamental constraint of the technique; the mitigation works by throttling all but the critical application. As soon as there are multiple critical applications participating in congestion to impacting real-time performance, this technique can not robustly mitigate.
- **Independent hardware interference is not mitigated**. The technique can only reduce interference that transitively is caused by software. If hardware peripherals inherently cause interference (i.e. a network interface controller triggers a Direct Memory Access (DMA) transfer to main memory due to an incoming Ethernet frame), our technique is incapable of mitigating the effects of that.
- **StressNG [Ki25] represents a worst case intruder**. StressNG [Ki25] is designed and utilized in the demonstrator to intentionally trash the caches. This is very unrealistic for most embedded applications, especially heavily controlled and scrutinized systems like avionics. In this demonstrator this leads to most multicore advantages being eliminated. In order to evaluate the realistic efficiency other, more controllable, intruder tasks are necessary.
- **The Wasm task used in the demonstrator is very basic**. The Wasm task that is used in this demonstrator consists only out of large matrix multiplications. While theses matrix multiplications are sensitive to interference, they do not represent a realistic task, which would be much more diverse in instruction and access patterns.
- **The demonstration relies on Linux specifics**. To gather more deterministic data on the effectiveness of the technique, a bare-metal implementation with as few moving parts as possible is desirable.

7 Outlook

This paper proposes an alternative approach to reduce the effects of multicore interference in safety-critical systems. While it is not possible to fully circumvent multicore interference in COTS hardware in general, it might be possible to dynamically limit the effects of interference while still preserving some of the multicore advantages. This demonstrator shows that the concept in itself is sound, but it does not show how viable this approach actually is, yet. For future experiments the demonstrator needs to be ported into a bare-metal setup, to get a more accurate testing environment and more relatable test results. To prove the viability of this approach it might also be necessary to swap the tasks for real workloads, e.g. avionics functions, to simulate realistic interference channels and amounts. If this approach should be employed in safety-critical systems the big question of certification has to be addressed. While this approach may not be usable in hard real-time systems it likely caters well enough for soft real-time system, which would require a statistical model of this approach to be derived and proved.

On the other hand it has to be mentioned that this approach of mitigating the problem of multicore interference — which is a problem rooted in the hardware architecture — in software, might not be the best solution because it adds additional complexity. Proper hardware that resolves all interference channels from the beginning would be more desirable, such as tailored manycore architectures. But for as long as there is an incentive to use COTS hardware, our presented approach may complement the state of the art as a viable alternative.

References

[22] WebAssembly Core Specification, version 2.0, W3C, 2022, https://www.w3.org/TR/wasm-core-2/.

[Co25] Costa, D. et al.: SP-IMPact: A Framework for Static Partitioning Interference Mitigation and Performance Analysis. In (Yomsi, P. M.; Wildermann, S., eds.): Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025). Vol. 128. Open Access Series in Informatics (OASIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:15, 2025, https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.NG-RES.2025.5.

[Ge26a] German Aerospace Center, I. o. F. S.: DLR-FT/dynamic-multicore-interference-mitigation, https://github.com/DLR-FT/dynamic-multicore-interference-mitigation, 2026, https://github.com/DLR-FT/dynamic-multicore-interference-mitigation.

[Ge26b] German Aerospace Center, I. o. F. S.: DLR-FT/wasm-interpreter, https://github.com/DLR-FT/wasm-interpreter/, 2026, https://github.com/DLR-FT/wasm-interpreter/.

[Ki25] King, C. I.: ColinIanKing/stress-ng, https://github.com/ColinIanKing/stress-ng, 2025, https://github.com/ColinIanKing/stress-ng.

[Lu22] Lugo, T. et al.: A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. IEEE Access 10, pp. 21853–21882, 2022.

[Ti22] Titzer, B. L.: A fast in-place interpreter for WebAssembly. Proc. ACM Program. Lang. 6 (OOPSLA2), 2022, https://doi.org/10.1145/3563311.

[Za23] Zaeske, W. et al.: WebAssembly in Avionics : Decoupling Software from Hardware. In: 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC). Pp. 1–10, 2023.

[Za25] Zaeske, W. et al.: On the Design of a WebAssembly Interpreter for Safety Critical Avionics Applications. In: 2025 AIAA DATC/IEEE 44th Digital Avionics Systems Conference (DASC). Pp. 1–10, 2025.

[ZH18] Zlatanchev, I.; Heidsieck, T.: Method and apparatus for executing real-time tasks, European pat. EP3296867A1, ESG Elektroniksystem und Logistik GmbH, 2018, https://worldwide.espacenet.com/patent/search?q=pn%3DEP3296867A1.

A Frequently Asked Questions

- **How does the scheduler described in this paper work?** We are not implementing a scheduler in this paper, nor are we using one. This (early) research is only concerned about mitigating interference between simultaneously running tasks on different CPU cores. A traditional scheduler could be employed on top of that.
- **Why do we use Wasm and not some other interpreter or e.g. the Java Virtual Machine (Java Virtual Machine (JVM))?** Because Wasm is a compilation target, not an interpreted language or a runtime. This provides a lot of flexibility in the implementation and integration into safety-critical systems. Especially because of its main intended purpose for cloud and web applications, the Wasm spec implements very strict spatial isolation. Further, via LLVM C, C++, Rust and Ada can be compiled to Wasm, allowing for existing software (-engineers) to be used. [Za25]
- **Does the utilized Wasm interpreter meet the complete compatibility to the regular Wasm runtime?** The utilized Wasm interpreter [Ge26b] is tested against the main part of the Wasm-Spec [22] test suite (~140 test procedures, ~51k test cases) and currently passes 100% of them. Their might be requirements that are currently not covered by the Wasm-Spec test suite, and there are probably some soundness issues in the implementation because it is not formally verified, but we are currently not aware of any Wasm binary that the interpreter cannot not run. The implementation of the interpreter does not include an implementation of the WebAssembly System Interface (WASI) Specification, which is a separate Specification for host OS functionalities like stdio, file system access, etc.
- **How would certification of the Wasm interpreter work? Given that it takes a lot of effort to verify software?** Yes, the Wasm interpreter would have to be qualified for certification, which is of course costly and commercially risky, but Wasm could also provide a multitude of other advantages to safety-critical systems [Za23], that could make certification worthwhile. Further, large effort is only required once per processor architecture instead of once per each application. A detailed discussion of certification friendly design for interpreters is laid out in [Za25].
- **Where can i find the source code?** On the GitHub page of the Institute of Flight Systems (DLR-FT): [Ge26a]