The Safety-Critical Edge Should Be Sparse

Liam McSherry
Department of Physics, University of Oxford
Oxford, United Kingdom
liam.mcsherry@physics.ox.ac.uk

Neil E. Bowles
Department of Physics, University of Oxford
Oxford, United Kingdom
neil.bowles@physics.ox.ac.uk

Abstract

The modern world is increasingly computerised and, as more functionality is shifted to the edges of systems in bids to reduce latency or energy consumption, the Internet of Things (IoT) increasingly includes safety-critical systems. A typical such system is 'functionally dense,' with a time-shared processing element (PE) carrying out many tasks; we believe that they should instead be functionally sparse, with many small PEs dedicated to single tasks. We present case studies of this design approach and introduce Funk, our effort to build the tools that will support the development of hard real-time, functionally safe systems that follow this approach.

CCS Concepts

• **Computer systems organization** → **Real-time systems**; • **Hardware** → **Safety critical systems**.

Keywords

functional safety, real-time computing, static analysis, RISC-V

1 Introduction

1.1 Reaching the edge

Traditional approaches to computing, at least as they existed from the creation of the first recognisably modern computer in *Colossus* in the 1940s, have been centralised. The precise shapes of these approaches have varied but, until relatively recently, the fact of users relying on a single central system has remained. For *Colossus*, cryptanalysts gave prepared programs to operators who would load them into the ten computers and distribute the results; with the introduction of time-shared mainframes in the 1960s, a user's access to computing would be through a terminal that did no computing of its own; the first modern computer networks of the 1970s and 1980s made time-shared mainframes accessible at a greater distance; and, throughout the 1990s to the 2000s, in general, a need for non-trivial computing power would be met by a server provisioned remotely in the 'cloud,' in a data centre, with users reaching out from desktop computers to ask that server for anything they might need.

The catalyst for a change in approach was the miniaturisation of computers in the 2000s and 2010s. Once a computer is small enough that one will fit in your pocket and move around with you, network access is both a significant drain on battery life (a Wi-Fi transceiver is only ≈45% efficient [4, 14]) and significantly less reliable. This, as well as the profit incentive of having the user do for free what you have been doing in costly data centres, helped drive the development of ever smaller and ever more capable computers in devices like mobile phones so that more could be done locally with the support of (rather than dependence on) the centre—that is, it helped drive the development of computing at the 'edge' of the network. As edge nodes become small enough and cheap enough to build into machines and appliances, their role changes too: having first been a means to access a single central resource, and then having been delegated computing tasks to carry out on behalf of this centre, these edge nodes now (as Haller *et al.* [9] put it) become active participants which can be queried, which can operate and co-operate autonomously, which can influence the physical world, and which can react in a way that is adapted to local conditions. It is from this change that the Internet of Things (IoT) emerges.

1.2 Repositioning the centre

Although it is tempting to frame IoT's impact in terms of the average person's day-to-day activities—in terms of smart thermostats and lightbulbs and of machine assistants that play music or order groceries—the most significant and widespread IoT applications will be in operational technology, construed broadly. The more obvious inclusions in this category range from sensors in lampposts to so-called 'vehicle-to-everything' (V2X) technologies and industrial control systems in factories, the electrical grid, and the water supply. Less obviously, the increasing trend towards software-defined platforms (that is, platforms that are reconfigurable in the field and often remotely) means that IoT now broadly includes mobile phone base stations, spacecraft, and the powertrain management systems in cars, among other safety-critical systems [1, 17, 19].

An attribute many of these applications share is their reliance on real-time systems to hold to hard deadlines. By definition, missing a hard deadline is a catastrophic failure of the system; contrastingly, missing a soft deadline only degrades quality of service. A circuit breaker failing to open in time can cause fires and damage to property while a TV sometimes failing to render a frame in ≈33.3 ms only gives a slightly worse viewing experience. Key in ensuring that a system is functionally safe is verifying that it meets its hard deadlines, but this is becoming ever more difficult as processing elements (PEs) play increasingly sophisticated roles and as the often singular PE at the edge is assigned more responsibilities—that is, as the PE at the edge becomes 'functionally dense' and begins itself to be a local centre in a decentralised system.

There are advantages to a functionally dense system. Most operating systems have multitasking abilities, so much of the tooling and knowledge around multi-responsibility systems will have been built in the context of one (logical) PE carrying out several tasks. Most mainstream workloads, too, have tended to benefit little from parallelisation, so until recently general-purpose PEs have become faster without becoming multiplicitous. For the hardware engineer, singular PEs mean fewer components and broadly simpler hardware and, similarly, for the assurance engineer they mean fewer configuration items to manage.

The downsides to a functionally dense system, however, are that many of these advantages work against the functional safety of the system. Most simply, with a singular PE the 'blast radius' of a

failure—the scope of the cascading failures it might cause—includes the whole of the PE's responsibilities, making much more complex the task of ensuring and proving what ISO 26262 [6] calls freedom from interference. Memory protection can limit blast radius but complicates the design of the PE, the software that runs on it, and verification outside of a live system. Secondly, the performance features that allowed PEs to become so fast did so at the expense of determinism—features such as caching, branch prediction, speculative execution, register renaming, dynamic frequency scaling, and out-of-order execution mean that the time a PE takes to execute a given routine is not only difficult to predict but might vary from run to run with the dynamic conditions the system experiences and in a way that cannot easily be bounded [11, 15]. Thirdly, contemporary optimising compilers used in building these systems present a double issue: they optimise for average-case performance rather than consistency and, consequently, their output may not be 'real-time stable,' requiring the software engineer to find the right incantation of source code to coerce them into producing well-timed machine code. Even the relaxed case of constant-time programming has only recently seen mainstream support [2]. This, like nondeterminism in execution time, is hard to avoid in functionally dense systems because they must be as fast as possible to service all their responsibilities while meeting their deadlines.

1.3 In pursuit of sparsity

The antipode of functional density is functional sparsity, where a singular PE with many tasks is decomposed into proliferated PEs each with few. Broadly, we see both high-level and low-level tasks migrating away from singular PEs; in a system-on-chip (SoC), now common in edge IoT devices, we see tasks shifted from 'big cores' to small and several ones, while within peripherals and subsystems what before might have been fixed-function finite state machines (FSMs) we see lifted into programmable PEs. This is due to the benefits to applying functional sparsity across the system hierarchy.

Perhaps counter-intuitively, one of these benefits is simplification. The physical partitioning of PEs encourages logical partitioning in their software and alignment with the commonly accepted principles of separating concerns, of limiting interdependencies in scope and motivation, and of abstracting services from their implementations. In hardware, because each PE is doing less, the PEs can be both behaviourally simpler—avoiding complex features like out-of-order execution—and electrically simpler by avoiding optimisations needed to reach higher clock frequencies, many-ported register files needed for instruction-level parallelism, and leading-edge process nodes needed to hit performance or area goals.

Following on from this, another benefit is that a smaller PE consumes significantly less energy, which is desirable both environmentally and, in an edge IoT context especially, to extend battery life or allow battery-free operation. Register files, for example, are often the largest energy consumer in a PE and their power consumption scales superlinearly with both number of ports and clock frequency [10, 20, 21]; superscalar, out-of-order, multiple-issue PEs need many-ported register files to avoid stalling and, with tens of ports and gigahertz clocks, can dissipate watts in their register files alone. A simpler design with a fewer-ported register file could need orders of magnitude less energy. Similarly, without the overheads of complicated control logic, large operating systems, and on-line scheduling, it could be imagined that a '1 GHz singular PE' might be replaced with four '200 MHz small PEs' and give superlinear improvements in otherwise linear-scaling losses from, for example, CMOS transistor switching or clock tree buffering. Outside the PE itself, reducing clock frequency and program size allows superlinear reduction in the power consumed by instruction memory [3].

Aside from these, functional sparsity has benefits more specific to hard real-time systems and functionally safe systems. For one, the physical partitioning between functions greatly limits the blast radius of a failure and simplifies freedom-from-interference analysis; broadly, failures in a functionally sparse system cascade only through explicit interfaces without a 'hidden' single point of failure in a shared singular PE. Similarly, static analysis of programs becomes easier as each program is simpler and program-to-program interactions can be analysed with less context at once—through those interfaces and the timing of events on them, rather than with the context of the whole program and a complex PE pipeline. With triple modular redundancy (TMR), area pressure on a simpler PE is lower and so trade-offs around TMR are easier to make. Relatedly, in multicore lockstep configurations the benefits are multiple: if hot sparing is used, simpler programs mean more opportunities to recover a failed replica PE; simpler PEs mean smaller and more reliable cross-comparison logic; and smaller PEs again reduce area pressure and, so, might allow use of triple-core instead of dual-core lockstep, improving reliability. The option to run at lower clock frequencies also improves reliability by decreasing susceptibility to single-event transients [7]. On the programmatics side, each simpler PE and its simpler programs will have correspondingly fewer requirements and smaller verification suites, which can significantly reduce the burden from high-assurance processes such as for ISO 26262 'ASIL D' functions or those in the higher criticality categories of the ECSS (European Cooperation for Space Standardization) ecosystem [6, 16].

Functional sparsity is not without downsides: it takes more effort to partition systems in the ways needed and, at present, tool support is lacking both for development under hard real-time constraints and under functionally sparse partitioning. We believe, however, that these are solvable problems, and the case studies we provide in sections 2 and 3 show how this approach might be followed and offer lessons for building the tools to build the tools.

2 Case study: The Modular Infrared Molecules and Ices Sensor on ESA's *Comet Interceptor*

2.1 Background

Comet Interceptor is the first European Space Agency (ESA) Fast-class mission [18]. Its goal is the first in situ observation of a long-period dynamically new comet—that is, a comet ejected from the Solar system early in its formation and which has never before transited the inner Solar system. As this comet returns to the Solar system, three spacecraft will wait at the Earth–Sun L_2 point. From there, mission planners can more promptly manoeuvre the spacecraft onto an intercept trajectory, before Solar heating and wind can alter the comet's surface and the pristine chemical building blocks it carries. Unaltered, these offer a time capsule view billions of years into the past. At encounter, the three spacecraft will fly in

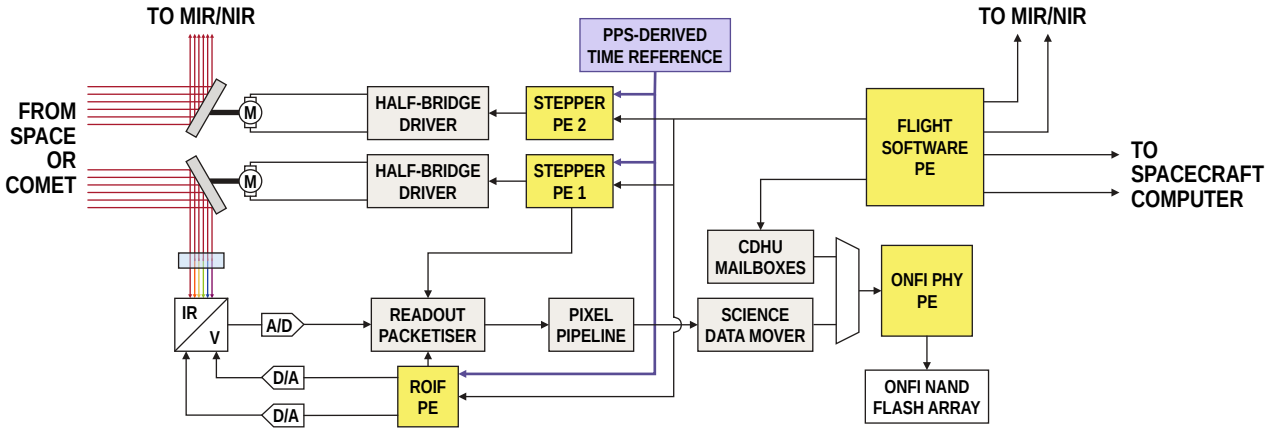

Figure 1: A schematic of MIRMIS' TIRI segment and CDHU, highlighting in yellow the individual processing elements (PEs): two for the stepper motors; one for TIRI's detector readout interface (ROIF); one implementing an Open NAND Flash Interface physical layer (ONFI PHY); and one executing the main flight software.

tandem past the comet with relative velocities of ≈70 km/s and a closest approach distance of ≈1000 km, giving the instruments on board as little as seven minutes to collect data. Selected by ESA in 2019, *Comet Interceptor* will launch in mid-2028 to early 2029.

The Modular Infrared Molecules and Ices Sensor (MIRMIS) is a hyperspectral sensor that will fly on *Comet Interceptor*'s main spacecraft and which is being built by the University of Oxford and VTT Finland. It is sensitive over 0.9–25 µm and has three segments: a near-infrared imager (NIR), a mid-infrared point spectrometer (MIR), and a thermal infrared filter radiometer (TIRI). TIRI is an integral part of MIRMIS' command and data-handling unit (CDHU), which interfaces to the spacecraft and manages the instrument.

2.2 Architecture

Shown in figure 1, the main functions of TIRI and the CDHU are divided between several PEs. The flight software (FSW) PE is a rad-hard SAMRH71 microcontroller, while the other PEs are hosted in a rad-tolerant ProASIC3 FPGA. All run at 50 MHz.

The stepper and detector readout interface (ROIF) PEs are identical and use a timing reference derived from a spacecraft pulse-per-second (PPS) signal to schedule instructions with 1 µs timing accuracy at intervals between 1 µs and ≈2.1 s. Each stepper PE controls a bipolar half-bridge driving a stepper motor, with one motor steering TIRI's pointing mirror and the other steering a shared MIR/NIR pointing mirror. These PEs get instruction lists from the FSW PE based on trajectory data from the spacecraft. The stepper PEs can apply arbitrary acceleration curves, minimising vibration, wear, and skipped steps while continuously tracking the comet, and can sample mirror position from an encoder with well known timing for feedback into a pointing control loop. The ROIF PE runs synchronously with the stepper PEs but instead interfaces with TIRI's detector, allowing fast rebiasing and rewindowing as well as triggering image captures at precise pointing angles and shorter intervals. Windowing the detector in real time—that is, reading out only select regions—increases returned science data by 11.5× and would be infeasible without hard-real-time control: TIRI's field of view is segmented by wavelength and must be swept back and forth over the comet; poor timing control could lead TIRI to miss the comet entirely. Faster captures and knowledge of position also reduce blurring and aid goniometry.

The Open NAND Flash Interface physical layer (ONFI PHY) PE is a pipelined VLIW core that signals on ONFI control and data buses with 10 ns timing precision to operate an 8 Gbit NAND flash array. This timing performance means the PE can maximise data write rate (and, hence, science return), but the main benefits are in risk reduction and assurance: development on the science data mover and FSW could be decoupled despite sharing resources, any radiation-induced degradation of the flash can be compensated for, and any defects discovered during development or in flight can be iterated on quickly and without reprogramming the FPGA (which is not possible in flight). On data rate, TIRI's detector continuously outputs ≈180 Mbps, but the flash guarantees only ≈140 Mbps writes and overheads (from packet metadata and error correction) inflate science data by ≈2.87×, meaning any lost performance has outsized impact on science return. In early testing, the PE reached ≈89–97% of maximum throughput depending on window dimensions.

2.3 Evaluation

The partitioning of tasks from the FSW PEs into the four smaller PEs was beneficial. It enabled performance not otherwise achievable, supported iterative development, and simplified the FSW. However, the smaller PEs were insufficiently independent of the FSW PE, in places adding more complexity than they removed.

Absolute-time synchronisation was a challenge. Stepper and ROIF PEs need 1 µs-accurate timing, but the spacecraft PPS cannot provide this. MIRMIS is asynchronous to PPS and so error cannot be averaged out as it can with a PPS-disciplined oscillator. This means PPS synchronisation happens once, immediately before encounter so as to minimise drift. Systems and tooling must be able to account for drifts like this to give safety guarantees. In addition, as trajectory calculation stayed on the FSW PE, transferring instructions in time could have become a limit; a PE capable of its own trajectory

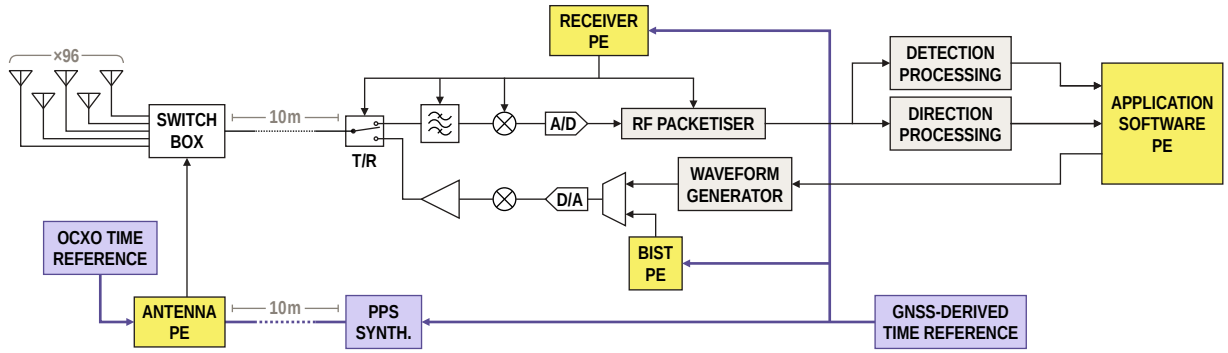

Figure 2: A schematic of CORVUS RAVEN, highlighting individual processing elements (PEs) in yellow: one controlling a passive electronically scanned antenna array; one controlling the radiofrequency frontend in a software-defined radio; one providing built-in self-test (BIST) functions; and one executing the main application software.

calculations would have been better, albeit less simple. These PEs have an area of ≈135 LUT3/FFs and consist of ≈500 lines of VHDL.

The ONFI PHY PE was influenced by a lesson learned from Lunar Thermal Mapper [5], where correct operation using a SAMV71RT microcontroller with a smaller detector had required manual cycle-counting and ignoring the spacecraft during readout. This and the ROIF PE together mean that almost all image-capture activity is delegated from the FSW PE. The challenges with this PE were accurately and consistently accounting for nanosecond delays through the system and scheduling the various parallel signalling actions, some of which we automated but much of which was manual. In testing, a regrettably large amount of test-by-inspection was needed, too. A tool for reusably expressing these delays, applying them, and with automated scheduling would have both decreased effort and increased confidence. Separately, for PEs such as these careful design of the instruction set is needed, and we found that differences between our predicted and actual needs made some routines more verbose (and so less performant) than they could have been. This PE's area is ≈210 LUT3/FFs from ≈900 lines of VHDL; its microcode has 461 instruction bundles across 15 subroutines.

3 Case study: CORVUS RAVEN counter-UAS localisation and inhibition in an SDR

3.1 Background

Recent global conflicts have proven the military effectiveness at very low cost of small uncrewed aerial systems (UAS), or drones, leading nations to invest in similarly low-cost and generally 'non-kinetic' counter-UAS technologies [12]. Often, these countermeasures are radiofrequency (RF) 'jammers' that transmit at high power to disrupt the data-links between a UAS and its ground control station, aiming to make the UAS uncontrollable, immobile, or otherwise ineffective. These counter-UAS jammers are often deployed as software-defined radios (SDRs) that operate across broad spectral bands, must interoperate with diverse equipment and hunt rapidly developing signals, and control high-power RF transmitters that pose a radiation hazard. They must do all of this in scenarios where inadequate response time, failure to detect a target, or giving incorrect information can lead directly to loss of life or materiel.

CORVUS RAVEN is a counter-UAS RF direction-finding and jamming system built by L3Harris [13]. It uses a switched-beam antenna, a kind of passive electronically scanned array (PESA), to fix the angle of arrival of RF emissions across 1.3–5.9 GHz, depending on antenna, and transmit inhibition signals in response.

3.2 Architecture

Figure 2 shows a schematic of RAVEN's main functions and their allocation between PEs. The receiver and built-in self-test (BIST) PEs are hosted in a Kintex UltraScale FPGA and run at 125 MHz, the antenna PE is hosted in a Zynq-7000 SoC at 125 MHz, and the application software (APP) PE is a 1.8 GHz i.MX8MQ processor.

The receiver and BIST use a 1 µs timing signal generated from a global navigation satellite system (GNSS)-disciplined clock. They and the antenna PE are bespoke 32-bit RISC cores, each with an area of ≈750 LUT6/FFs, running programs known as timeplans. RAVEN uses the receiver PE to frequency-hop by rapidly retuning the RF frontend, to operate a transmit–receive (T/R) switch, and to trigger the capture of I/Q samples from the transceiver into VITA-49 packets. The PE does this synchronously with the antenna and BIST PEs, with the BIST PE periodically transmitting test signals through the frontend for on-line antenna tuning and health-checking.

The antenna PE is remote from the other PEs and sits inside the PESA, where it switches in and out subsets of 96 antenna elements to form a beam steerable in azimuth and elevation. The PE moves the beam every ≈40 µs in a random pattern intended to increase probability of intercept, and it does this synchronously with the receiver PE so that the beam is stable when samples are captured and so that the receiver PE can tag samples with pointing information to feed to the algorithms implemented by the APP PE. This synchronisation is possible because of three design choices: the antenna PE has a 1 ppb ovenised oscillator (OCXO) that limits phase drift to <1 µs/s after ≈12 hours of GNSS holdover; the receiver and antenna PEs share GNSS-derived PPS that they use to periodically cancel phase between them; and RAVEN's timeplans are designed around milestones on a 1 µs raster. In combination, this ensures the PEs are always on the same milestone.

The APP PE is asynchronous to the other PEs and interacts with them through sets of mailboxes, receiving and processing VITA-49

packets captured on the receiver PE's trigger. As well as implementing signal detection and direction-finding algorithms, the APP PE also deals with networked command-and-control messaging, connections to end-user devices, and system set-up and configuration. The APP PE runs a customised Linux distribution.

3.3 Evaluation

In RAVEN, hard-real-time PEs are essential to meet the system's timing goals. The main benefit is the distributed synchronisation of the PEs, which allows complex operation of the PESA across form factors—a vehicle-mounted radio, for example, can remain at the bottom of a ≈10 m mast, while a manpack radio can be used without bulky many-way connectors. Locally, they allow sample-level timing of actions, which increases performance by minimising guard durations without risking damage through, for example, a high-power transmission directed into the isolated transmit port of the T/R switch. Similarly, on safety, with equivalent isotropically radiated powers of ≈67 dBm in a ≈25° beam, coordinated and deterministic responses are mandatory for antifratricide operation and to minimise the radiation hazard to end-users.

Verifying RAVEN's timeplans was a significant challenge. The three receiver, BIST, and antenna timeplans had to be aligned with one another, which was a manual task and so vulnerable to mistakes and configuration management issues. The lack of tooling to support this limited performance—RAVEN's main antenna has 96 elements, but smaller 48-element antennas were also tested; although, in theory, the coarser-grained beams could be swept twice as fast, in practice manually maintaining many diverse timeplans for different antennas was seen as too risky and, instead, a 'one size fits all' template was used with the slack filled by busy-waits. A similar issue was present with frequency hopping, where timeplans would need to be manually tweaked for different numbers of bands of interest. This slowed development as functional changes to timeplans needed manual recalculation of delays and guard durations. As a further consequence, on what was otherwise a user-configurable SDR only the factory could create new RAVEN system configurations, negatively affecting end-user experience and lengthening response time to new battlefield threats.

Similarly to MIRMIS, reusably modelling and applying delays was an issue, and it was an issue complicated by the asynchronicity between the antenna and receiver PEs that meant the timing between them continuously changed and that they would observe the notionally-singular shared PPS pulse at different times.

Timeplans for CORVUS SDRs were not normally written by hand; rather, they were assembled from a manifest that listed which pre-timed blocks to insert when, and the timings stated for these blocks would be summed to check that a limit was not exceeded and that extra delays were inserted between blocks if needed. This worked well for simpler and 'straight line' timeplans, like early BIST timeplans, which repeatedly carried out the same set of actions, but it ran into issues when more functionality was partitioned off from the APP PE. At a high level, this tool did 'forwards' analysis—it could indicate whether a given manifest exceeded a limit, but it could not work 'backwards' to, say, fill a duration with as many repetitions as possible. It was also unable to rearrange code across block boundaries, insert loops, or deal with complex branching code, leading either to unrealistically pessimistic timing estimates or, like in RAVEN's case, to manually maintained timeplans. Additionally, it could not assist with cases where multiple timeplans needed to be synchronised (as for the antenna, receiver, and BIST PEs), in some cases working against the engineer as it arranged pre-timed blocks without knowledge of a synchronisation requirement.

4 Bringing the Funk

4.1 Now, what's the problem?

The common theme with many of the issues noted in the case studies is tooling—as more complex functions are partitioned off, manually keeping track of timing concerns, consistently and correctly taking them into account, verifying that constraints are met, and validating those constraints becomes unwieldy, and where automation does exist its separation from the semantics of the code can make the tool an obstacle itself. This in mind, we introduce Funk—our effort to build tooling for these use-cases that offers an ergonomic way to manage timing concerns and which is assured in line with functional safety standards, based on work done for MIRMIS' ONFI PHY and now funded through 2027.

4.2 Just bring 'em some Funk

Our effort is split into two threads: building a compiler that can manage these timing concerns, and building a deterministic processor core this compiler can target. We will release both under open-source licences on Codeberg[1] when ready.

Our intention is for the core, a RISC-V core called `FV32A`, to be a minimal useful target. This makes our path to a working compiler simpler than if we were to target existing timing-predictable cores with more complex pipelines, such as [8], and means we will have a target more suited to our deeply embedded use-cases. `FV32A` is designed as a scalar, in-order, RV32E core working from a single-ported tightly coupled memory, with a target of <200 ns/instruction at 50 MHz on common space-qualified FPGAs. We are including high-reliability features including error-corrected memory, a 'trace/lockstep interface' to support debugging and multicore lockstep configurations, support for RISC-V control flow integrity extensions, and modified interrupt handling to better support working with timing reference signals. We aim to support dual qualification to ECSS criticality category A and as an ISO 26262 safety element out of context with ASIL D capability.

For the compiler, we are similarly aiming for dual tool qualification. We are beginning development with the 'middle end' because it is where the bulk of the problem solving must happen, with the frontend notionally intended to implement a LISP dialect. This, we think, aligns well with the use-case: for example, for deterministic timing the compiler must ensure that programs are total (that is, prove they terminate), and so a powerful macro system such as those in LISPs will help regain lost expressiveness. That said, we intentionally leave the exact shape of the frontend as something currently unfixed and likely to see substantial change.

In the compiler's middle, we intend to structure programs around events, which can be connected to one another through timing relationships and to which there can be associated sections of logical

[1] https://codeberg.org/microcircuits

code. Events and relationships we intend as first-class types, manipulable within a macro expansion phase and referenceable within logical code, so that user-defined analysis and code generation is practical and responsive to the compiler's scheduling. As an initial target for this programme of work, these events will be relatively static (a timeline, for example, or offsets from a timing reference signal), but our long-term aim is for message-passing channels to be represented and for the compiler to determine event activity from static analysis. We think these structures will also provide an ergonomic way to model external devices, initially purely deterministically (such as accessing tightly coupled memory or a single-manager bus) but later with bounded nondeterminism (such as process actuators with well-defined fault tolerance time intervals). We also intend to take lessons learned from newer memory-safe languages and implement stricter ownership semantics where values must be consumed for the program to be valid.

This middle will feed a backend that implements not only code generation but also timing analysis, using models to calculate execution time, and scheduling where code is rearranged or has waits inserted to satisfy timing relationships.

4.3 It's not the end, but I can see it from here

As a first step towards maturing work from MIRMIS, we are starting conservatively and aiming to expand what the compiler can prove later. Assurance activities for the `FV32A` core are ongoing with active development to begin shortly; for the Funk compiler, the definition of the concept of employment/CONEMP is ongoing with assurance and development to begin after development on `FV32A` concludes, with MIRMIS-derived tooling as a starting point. We aim to have a useful demonstrator of the compiler by mid 2027.

Once we have a working demonstrator, our aim is to build smarter control flow analysis, so that FSMs and truly event-driven systems can be represented. We want to allow analysis across foreign function interfaces, where possible. We want to build feedback-driven optimisation between code generation, timing analysis, and scheduling. We want to explore automated parallelisation, and target more performant processors that maintain determinism, including MINOTAuR [8] and others better suited to deeply embedded applications. With these goals in mind, even if their implementation is far in the future, we hope to put the right scaffolding in place so that supporting them needs a minimum of change.

Acknowledgments

This work has received support from several sources, most recently through the NGI Zero Commons Fund established by NLnet with financial support from the European Commission and the Swiss Secretariat for Education, Research and Innovation (SERI).

The University of Oxford's contributions to *Comet Interceptor* were supported by the UK Space Agency's Space Science Programme. We also thank T. Szymkowiak, P. Madle, A. Cowling, B. Hudson, K. Mitchell, and G. Quelcuti for their contributions.

The development of RAVEN was funded by and carried out at L3Harris Technologies and we thank A. A. Beard, T. Harriss, H. Austen, P. Abbey, O. Jeffreys, M. Ainsworth, G. Hudson, and the wider CORVUS team for their work.

References

[1] European Space Agency. 2021. European software-defined satellite ready to start service. [Online]. Available: https://www.esa.int/Applications/Connectivity_and_Secure_Communications/European_software-defined_satellite_ready_to_start_service. Accessed: 20-Jun-2026.

[2] Julius Alexandre. 2025. Introducing constant-time support for LLVM to protect cryptographic code. *The Trail of Bits Blog* (Dec. 2025). https://blog.trailofbits.com/2025/12/02/introducing-constant-time-support-for-llvm-to-protect-cryptographic-code/

[3] B.S. Amrutur and M.A. Horowitz. 2000. Speed and power scaling of SRAM's. *IEEE Journal of Solid-State Circuits* 35, 2 (2000), 175–185. doi:10.1109/4.823443

[4] Arul Balasubramaniyan, Abdellatif Bellaouar, Dominik Martin Kleimaier, Miguel Meza Campos, and Shafi Syed. 2025. A 22FDX® Wi-Fi PA Achieving Output Power of 1.1Watts with 41% PAE at 3.8V Supply Voltage. In *2025 IEEE European Solid-State Electronics Research Conference*. Munich, DE. doi:10.1109/ESSERC66193.2025.11213977

[5] Neil E. Bowles et al. 2026. The Lunar Trailblazer Lunar Thermal Mapper Instrument. *Journal of Geophysical Research: Planets* 131, 5 (2026). doi:10.1029/2025JE009333

[6] International Organization for Standardization. 2018. *Road vehicles — Functional safety — Part 1: Vocabulary* (2nd ed.). Standard ISO 26262-1:2018(E). Geneva, CH.

[7] M.J. Gadlage, R.D. Schrimpf, J.M. Benedetto, P.H. Eaton, D.G. Mavis, M. Sibley, K. Avery, and T.L. Turflinger. 2004. Single event transient pulse widths in digital microcircuits. *IEEE Transactions on Nuclear Science* 51, 6 (2004), 3285–3290. doi:10.1109/TNS.2004.839174

[8] Alban Gruin, Thomas Carle, Hugues Cassé, and Christine Rochange. 2021. Speculative Execution and Timing Predictability in an Open Source RISC-V Core. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. 393–404. doi:10.1109/RTSS52674.2021.00043

[9] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. 2009. The Internet of Things in an Enterprise Context. In *Future Internet – FIS 2008*, John Domingue, Dieter Fensel, and Paolo Traverso (Eds.). Springer, Berlin, DE, 14–28.

[10] Nam Sung Kim and Trevor Mudge. 2003. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 172–182. doi:10.1145/782814.782839

[11] Raimund Kirner and Peter Puschner. 2008. Obstacles in Worst-Case Execution Time Analysis. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 333–339. doi:10.1109/ISORC.2008.65

[12] Chris Kremidas-Courtney. 2026. *The new economics of warfare*. European Policy Centre. https://www.epc.eu/publication/the-new-economics-of-warfare/

[13] L3Harris Technologies 2025. L3Harris showcases counter-drone capability to British soldiers at VANAHEIM. https://www.l3harris.com/newsroom/editorial/2025/07/l3harris-showcases-counter-drone-capability-british-soldiers-vanaheim

[14] Guansheng Lv, Wenhua Chen, Long Chen, Fadhel M. Ghannouchi, and Zhenghe Feng. 2024. A 4.9–7.1-GHz High-Efficiency Post-Matching GaN Front-End Module for Wi-Fi 7 Application. *IEEE Journal of Solid-State Circuits* 59, 2 (Feb. 2024), 400–413. doi:10.1109/JSSC.2023.3288390

[15] Kelvin D. Nilsen and Bernt Rygg. 1995. Worst-case execution time analysis on modern processors. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems* (La Jolla, California, USA) *(LCTES '95)*. Association for Computing Machinery, New York, NY, USA, 20–30. doi:10.1145/216636.216650

[16] ESA Requirements and Standards Division. 2017. *Space product assurance — Dependability*. Standard ECSS-Q-ST-30C Rev.1. European Cooperation for Space Standardization.

[17] Dirk Slama, Achim Nonnenmacher, and Thomas Irawan. 2023. *The Software-Defined Vehicle* (1st ed.). O'Reilly Media, Sebastopol, CA, United States.

[18] C. Snodgrass and G. H Jones. 2019. The European Space Agency's Comet Interceptor lies in wait. *Nature Communications* 10, 5418 (2019). doi:10.1038/s41467-019-13470-1

[19] Giuseppe Tricomi, Carlo Scaffidi, Giovanni Merlino, Francesco Longo, Antonio Puliafito, and Salvatore Distefano. 2023. A Resilient Fire Protection System for Software-Defined Factories. *IEEE Internet of Things Journal* 10, 4 (2023), 3151–3164. doi:10.1109/JIOT.2021.3127387

[20] Xiaoyang Zeng, Yi Li, Yuejun Zhang, Shujie Tan, Jun Han, Xingxing Zhang, Zhang Zhang, Xu Cheng, Jun Han, and Zhiyi Yu. 2015. Design and Analysis of Highly Energy/Area-Efficient Multiported Register Files With Read Word-Line Sharing Strategy in 65-nm CMOS Process. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 7 (2015), 1365–1369. doi:10.1109/TVLSI.2014.2334693

[21] V. Zyuban and P. Kogge. 1998. The energy complexity of register files. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*. 305–310. doi:10.1145/280756.280943