Computers & Security 123 (2022) 102920

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

# Behind the last line of defense: Surviving SoC faults and intrusions

Inês Pinto Gouveia[a,*], Marcus Völp[a], Paulo Esteves-Verissimo[b]

[a] University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group, Luxembourg
[b] KAUST - King Abdullah University of Science and Technology, Resilient Computing and Cybersecurity Center (RC3), Saudi Arabia

ARTICLE INFO

*Article history:*
Received 9 December 2021
Revised 3 September 2022
Accepted 10 September 2022
Available online 13 September 2022

*Keywords:*
Fault and intrusion tolerance
Reliability
Hypervisor
Processor architecture
MPSoCs

ABSTRACT

Today, leveraging the enormous modular power, diversity and flexibility of manycore systems-on-a-chip (SoCs) requires careful orchestration of complex and heterogeneous resources, a task left to low-level software, e.g., hypervisors. In current architectures, this software forms a single point of failure and worthwhile target for attacks: once compromised, adversaries can gain access to all information and full control over the platform and the environment it controls. This article proposes *Midir*, an enhanced manycore architecture, effecting a paradigm shift from SoCs to distributed SoCs. *Midir* changes the way platform resources are controlled, by retrofitting tile-based fault containment through well known mechanisms, while securing low-overhead quorum-based consensus on all critical operations, in particular privilege management and, thus, management of containment domains. Allowing versatile redundancy management, *Midir* promotes resilience for all software levels, including at low level. We explain this architecture, its associated algorithms and hardware mechanisms and show, for the example of a Byzantine fault tolerant microhypervisor, that it outperforms the highly efficient MinBFT by one order of magnitude.



## 1. Introduction

Economic wealth and the well-being of modern societies depends on information and communication technologies (ICT). Such dependency obviously hinges on the correctness of these systems, some of them critical, which may fail in a combination of multiple causes and ways (Davies, 2016; Lee, 2018; Lee et al., 2016; Price, 2019; Tsidulko, 2018; Yusof, 2019). Systems have been progressively pushed to extremes of efficiency through modularity in platform sharing, firstly through virtualization and lately by leveraging the enormous power growth, functional diversity and adaptation flexibility offered by multi- and manycore architectures. This has taken platform sharing to new heights, into the realm of multiprocessor systems-on-a-chip (MPSoCs).

The organization of these complex computing resources depends on low-level platform management hardware (e.g., memory-management units (MMUs)) and software (e.g., firmware, hypervisors, management engines). However, current MPSoC architectures are such that these management components, which should form a last line of defense against severe accidental faults or adversaries intruding the system (malicious faults), instead constitute a single point of failure (*SPoF*), for two main reasons. First, the way platform privilege-enforcement mechanisms (e.g., MMUs or hardware-enforced capabilities (Woodruff et al., 2014)) are designed allows faults in a core/tile to propagate through MPSoC components. Second, faults in this lowest-level management software, e.g., hypervisors configuring these privileges, are bound to propagate across management and managed components, again causing common-mode failure scenarios.

If these SPoFs are compromised by adversaries, the latter gain full authority over the platform's privilege-enforcement mechanisms and, through them, access to all information and complete control over all platform resources (e.g., cloud-based systems) (Szefer et al., 2011a), including, in the case of cyber-physical systems, extended control over the physical environments on which they act (e.g., nuclear power plants Das, 2019, power grid stations Meserve, 2007 or contemporary and autonomous cars Greenberg, 2015).

Is this a real risk? It is, if the vulnerability rate of these low-level platforms is non-negligible. Continuing problems, whether in Intel's CSME (Ermolov and Goryachy, 2017), Xen/Critix (Xen, 2019) or concerning Spectre (Kocher et al., 2018) and Meltdown (Lipp et al., 2018), have been repeatedly reminding us of how brittle the assumption of "tamperproof and unattackable low-level platform management assets" is. Numerous vulnerabilities have been reported in RTOSs' source code, namely in IoT devices (e.g., CWE-119, CWE-120, CWE-126, CWE-134, CWE-398, CWE-561, CWE-563) (Al-Boghdady et al., 2021). Vulnerability

* Corresponding author.
*E-mail addresses:* ines.gouveia@uni.lu (I.P. Gouveia), marcus.voelp@uni.lu (M. Völp), paulo.verissimo@kaust.edu.sa (P. Esteves-Verissimo).

https://doi.org/10.1016/j.cose.2022.102920

analysis of virtualized environments and hypervisor security have shown the various ways these can be attacked (Brooks et al., 2012; Prabahar and Edwin, 2012; Thongthua and Ngamsuriyaroj, 2016; Turnbull and Shropshire, 2013). Even formally verified kernels (e.g., seL4 Klein et al., 2009) may fail due to model/reality discrepancies or hardware faults violating modelling assumptions (Biggs et al., 2018).

Being the risk real, are there no solutions yet? The solution design space for contemporary hardware platforms dependability and security has been unfolding in two directions: (i) application-specific system-level replication (e.g., triple modular redundancy, mainly in cyber-physical systems (CPS), by means of multiple electronic control units (ECUs)), where the lack of flexibility limits the extension to general systems; (ii) manycore-level replica management and consolidation, which then, if on bare MPSoCs, reintroduces the SPoF concern, now for the low-level replication management component (Baumann et al., 2009; Bressoud and Schneider, 1995; Döbel, 2014; Esposito et al., 2018; Lamport, 1998).

At this time, we call the reader's attention to an interesting fact, which will become crucial to our solution. The current MPSoC architectures' complexity, modularity and networked inter-connectivity, suggests attributes of distributed systems (Mullender, 1993), albeit imperfect such systems (an example of which is the aforementioned SPoF syndrome). Heavily studied techniques have been used in distributed systems to mitigate SPoF syndromes and to implement fault and intrusion tolerance schemes (Powell et al., 1988; Verissimo et al., 2006), such as replication and consensus. In consequence, the root of the MPSoC problems just presented may also be an avenue to their solution, i.e., we can leverage the available resources in MPSoCs (e.g. cores) and their connectivity together with lessons from the distributed systems realm to solve the presented issues. This comparison of (MP)SoCs to distributed systems was first made in Függer and Schmid (2012), where a fault-tolerant clock generation mechanism for SoCs is introduced.

So, in this article, we start by identifying the gaps from (MP)SoCs to distributed systems and proposing (MP)SoC mechanisms to bridge them, which essentially means achieving: fault independence and fault containment, despite low software-level compromise, while retaining the flexibility (MP)SoCs offer. Having a manycore that behaves as a (closely-coupled) distributed system should allow us to design a set of efficient and low-overhead distributed systems-inspired modular protection and redundancy management mechanisms, e.g., Byzantine fault tolerant state machine replication (BFT-SMR), for fault and intrusion tolerance (FIT). The remaining problem, how to implement and where to locate all the mechanisms above, is addressed by the *Midir*[1] architecture presented in this paper, which leverages the computing critical mass and flexibility of contemporary tile-based manycore architectures.

*Midir* constrains the connection of all tiles to the network-on-chip (NoC) through simple, self-contained hardware-based trusted-trustworthy components, which we call *T2H2*. Exploring the concept of architectural hybridization (Veríssimo, 2006), whilst we consider those components to be ultra-reliable and not fail, we are agnostic about the reliability of individual tiles, which may be compromised or fail. The assumption is justified by the simplicity of the former, promoting verifiability.

The *T2H2* components implement the functionality required for fault independence, containment, and tolerance mechanisms mentioned above. In consequence, tile-internal software or hardware faults are contained in the tile and the objects the tile can access. Furthermore, the baseline mechanisms for protection and redundancy management provided by *T2H2* can be extended and recursively applied at any software layer, giving the designer ample latitude for crafting resilience into systems, both "horizontally" (incremental power of defense mechanisms) and "vertically" (depth of defense).

Locating *T2H2* between the tile and the NoC interconnect not only provides a clear pathway for integration by chip manufacturers and integrators, it also allows drawing from many well-understood building blocks (e.g., region protection, capabilities Needham and Wilkes, 1974, and other chip-level resource management mechanisms Aggarwal et al., 2007, capable of isolating tiles and the resources they can access). The novelty of *Midir* lies in their arrangement to avoid SPoFs, even while they are reconfigured.

The contributions of this article are:

1. An analysis of the gaps separating current MPSoC architectures from genuine distributed systems and how gap fixing, through measures promoting fault independence and fault containment in tile-based architectures enforced at the level of the tile-to-NoC interface, secures fault isolation and the elimination of SPoFs.
2. An architecture (*Midir*) leveraging the resulting distributed system-on-a-chip (DSoC) in (1) to achieve incremental levels of modular fault and intrusion tolerance, through a range of diverse redundancy management techniques implemented by simple hardware-based voting/consensus mechanisms.
3. The design of a simple and ultimately **t**rusted-**t**rustworthy **h**ardware **h**ybrid, *T2H2* — the core component of *Midir*, staged at the tile-to-NoC interface — providing just two generic baseline functions: access control (capability registers) and quorum-based consensus (voters). Through configurations and combinations of these two basic functions, *T2H2* is capable of implementing all the techniques mentioned in (1) and (2).
4. As a proof of concept, we give and evaluate an implementation featuring *Midir* and essential parts of a fault and intrusion tolerant microhypervisor built on top of it. Although the architecture serves several reliability strategies, we chose the most effective, active replication with error masking. Being the most complex and costlier, we believe to have shown the performance and practicality of our concept.

An analysis of related work is presented next (Section 2), followed by an evaluation of the challenges for bridging SoCs to DSoCs (Section 3), and the threat model (Section 4). Then, we introduce the *Midir* architecture (Section 5) and the *T2H2* component in Section 6. At this point, we are able to show *Midir* in action, discussing the design of a fault and intrusion tolerant microhypervisor built on top of it (Section 7), as an example of critical low-level management software. Finally, we discuss some relevant implementation matters in Sections 8 and 9, we evaluate *Midir* on a Zynq ZC702 board, showing how *Midir*'s hardware voters accelerate BFT-SMR protocols, voted execution of system calls and consensual reconfiguration of *T2H2*. We shall say an operation is *consensual* if it becomes effective only after a fault threshold-exceeding quorum of replicas agreed to executing this operation, here by means of voting. Section 10 concludes the paper, pointing to directions for future work.

## 2. Related work

In this Section, we present several classes of works that motivated *Midir*: low-level approaches for detection and containment of errors in low-level support software; analyses of the evolution of defects in system support software; attempts at preventing and/or mitigating the resulting errors and potential failures; approaches to replication-based fault/intrusion tolerance and resilience.

[1] pronounced meedir

Mitigation measures have been studied for detection and containment of errors in OS and manycore support software (Döbel, 2014; McCune et al., 2010; Seshadri et al., 2007) through an underlying, assumed-trustworthy layer. However, they still have a non-negligible complexity, and in consequence, even a residual fault or vulnerability rate in these supposedly trusted components may breach the platform's dependability and security goal.

In fact, as confirmed by Hoffmann et al. (2013), "simple" components with at least a few KLOCs have a non-negligible statistical fault footprint. Other studies (Ostrand and Weyuker, 2002; Ostrand et al., 2004) reveal between 1–16 bugs per 1000 lines of code go undetected before deployment, even in well-tested software, and operating-system kernels form no exception (Matias et al., 2014; Patterson and Ganapathi, 2005). Recent insights (Palix et al., 2014) reveal that faults in stateful core subsystems — on which we focus here — outrank driver bugs in severity.

Many approaches target operating systems with the goal of improving their resilience against faults. However, typically they protect either applications (Bolchini et al., 2013; Depoutovitch and Stumm, 2010; Kuvaiskii et al., 2016) or specific OS subsystems (Elphinstone and Shen, 2013; Sundararaman et al., 2010; Swift et al., 2006; Zhou et al., 2006) and only from accidental faults. Efforts for providing whole-OS fault tolerance include (Bhat et al., 2016; David et al., 2008; Gens, 2018; Govil et al., 1999; Herder et al., 2006; Lenharth et al., 2009; Nikolaev and Back, 2013). Furthermore, the complexity of these recovery kernels is comparable to that of a small hypervisor. For example, OSIRIS (Bhat et al., 2016) directs OS recovery to a 29 KLOC reliable computing base (RCB) (Engel and Döbel, 2012), roughly twice the size of modern microkernels (Asmussen et al., 2016; Klein et al., 2009; Lackorzynski et al., 2018; Liedtke, 1995). Again, this makes the likelihood of residual faults or vulnerabilities non-negligible.

Several other works have given early steps in the direction of the solutions we advocate in this paper, minimizing the threat surface, or enforcing isolation. Nohype (Szefer et al., 2011b) removes all but a small kernel substrate from application cores, which run functionality-rich OSs in virtual machines (VMs), reducing the threat surface. Cap (Needham and Wilkes, 1974) and M3 (Asmussen et al., 2016) exploit hardware capability units and Hive (Chapin et al., 1995) a bus-level firewall to isolate VMs at tile granularity. However, although this avoids trusting tile-local kernel substrates for isolation, their configuration interface, which is necessary to retain flexible resource sharing, turns configuring the kernel into a single point of failure. We address this problem in *Midir*, by requiring reconfiguration of the fault-isolating *T2H2* unit to be performed only if agreed by a majority of correct replicas operating in consensus.

Capabilities are cryptographically- (Tanenbaum and Kaashoek, 1994), kernel- (Hardy, 1985; Lackorzynski et al., 2018; Shapiro and Hardy, 2002) or hardware-protected tuples (Asmussen et al., 2016; Needham and Wilkes, 1974) comprised of at least a pointer to an object (or service) and access rights authorizing which operations owners of these capabilities may execute on the object. Posession of a capability is both necessary and sufficient to exercise a granted access over an object. Consequently, as long as both capability-enforcement and -reconfiguration are trustworthy, faults in a component cannot propagate beyond the objects it can access, unless other accessing components are faulty as well.

Cheri (Woodruff et al., 2014) adds capability protection on top of page-based protection, but includes the MMU and the OS page-table management in the reliable computing base (RCB), which means the former must be trustworthy. The concept behind *Midir* is independent of the protection model, and thus not necessarily tied to e.g., capabilities. Also, by establishing the fault containment domains at the granularity of tiles, we are agnostic about the semantics and interplay of tile-internal and/or core-level components, e.g., MMUs, memory protection or page-table management. Enforced by *T2H2*, the protection mechanisms are crafted at inter-tile level, emulating the spacial isolation of distributed system nodes.

Replication has been used before in closely-coupled systems, primarily to tolerate accidental faults in cyber-physical systems (CPS), by replicating controllers to form triple modular redundant (TMR) units, or duplicated self-checking units. An example of the use of TMR in highly critical systems can be seen in the primary flight computers of Boeing 777's fly-by-wire (FBW) system (Yeh, 1998). In a similar context, a form of passive redundancy can also be seen in Airbus' dependability-oriented approach to FBW, where "hot spares" are used in case the active computer interrupts its activity (Traverse et al., 2004). The concept was extended to multi-phase tightly synchronous message-passing protocols still in the CPS domain (Kopetz and Bauer, 2003; Mancini, 1986). The so-called 'Paxos' (Schiper et al., 2014), and 'Byzantine' (Castro and Liskov, 1999) Fault-Tolerant State-Machine Replication classes of protocols promote resilience to threats, respectively failures and both threats and failures, extending the concept to generic classes of applications, namely in loosely-coupled systems. For example, Castro's seminal BFT-SMR protocol (Castro and Liskov, 1999) masks the actions of a minority of up to $f$ compromised replicas, by reaching a majority voted consensus of $|Q| = 2f + 1$ out of $n = 3f + 1$ replicas. Behind all the categories of techniques above is a baseline voting mechanism among the values proposed by a pre-defined number of replicated fault-independent components. *Midir* offers such a baseline mechanism at a low enough level of abstraction to serve essentially any replication-oriented application.

Architectural hybridization (Veríssimo, 2006) (i.e., the inclusion of trusted-trustworthy components that follow a differentiated fault model) allows reducing $n$ and $|Q|$ to $2f + 1$ and $f + 1$, respectively (Correia et al., 2004; Kapitza et al., 2012; Levin et al., 2009; Veronese et al., 2013a). The implementation of *T2H2, Midir*'s trusted component, draws from these quorum reduction results, and further accelerates the BFT-SMR protocol that *Midir*-enabled FIT microhypervisors use to coordinate system call execution (Section 7).

Paxos and BFT replication have been attempted as well inside MPSoCs (Baumann et al., 2009; Bressoud and Schneider, 1995; Döbel, 2014; Esposito et al., 2018; Lamport, 1998). However, all these works were made under the assumption of a trusted low-level kernel (e.g., hypervisor or platform manager), which obviously is a single point of failure (SPoF). One of the key results of *Midir* lies in the realization of the distributed system-on-a-chip (DSoC) vision, which enables such replication management techniques in MPSoCs, whilst removing the SPoF syndrome of the low-level kernel.

Aguilera et al. (2020) leverage RDMA in the crash fault-tolerant system Mu to bring SMR performance down to microsecond scale, also for BFT (Aguilera et al., 2019). *Midir* aims at reaching consensus with a performance close to the speed of the NoC.

## 3. Gap analysis and system model: from MPSoCs to distributed MPSoCs

MPSoCs consolidate in a single chip computing resources that used to reside on multiple chips. Tiles (Waingold et al., 1997) are placeholders and instantiation points for resources, typically instantiated with cores and private caches, or with slices of shared caches and connected through the NoC with each other and with memory controllers (to reach out to RAM/IO). It is possible as well to cast accelerators, GPUs and FPGAs into the tile abstraction.

The modularity and networked interconnection of tiles already suggests attributes of a distributed system and has inspired first steps to hardware-enforced fault containment at tile level, as pi-

oneered by Hive (Chapin et al., 1995) and M3 (Asmussen et al., 2016). Tiles favour functional and non-functional diversity since they can host cores from several makers. This improves fault independence through the implied low likelihood of experiencing the same fault in different tiles. Similarly, different versions of the same code can be used at distinct tiles with the same intent (Avizienis et al., 1977; Joseph and Avizienis, 1988; Knight and Leveson, 1986). However, fault containment remains imperfect: potentially faulty or compromised low-level kernels retain control over platform privilege configuration mechanisms and, thus, form a single-point of failure.

In our system model, we therefore assume a fully connected tiled system, where on-chip network components offer the abstraction of a correct network, interconnecting all tiles to one another. Messages sent are eventually delivered, unchanged, to the destination, but possibly only after several retries. Network coding (Ogg et al., 2008), multi-tenant (Colman-Meixner et al., 2016) and adaptive routing techniques (Yang et al., 2016) increase the coverage of this assumption. We leave coverage of network attacks and their mitigation for future work.

We shall further assume tiles are instantiated with heterogeneous processing elements and will hence exhibit a certain level of fault independence through the implied low likelihood of experiencing the same fault in different tiles — diversity.

Conventional multi- and manycore designs retain the possibility of common mode failures in central hardware components (e.g., the clock or power distribution network), which must be addressed differently. Resilient clocks (Schmid and Steininger, 2010) mitigate some of these common-mode faults and the recent trend towards interconnected chiplets further improves the physical decoupling of tiles. Once physical (hardware) effects of a fault are retained to the causing tile and the signals it exhibits to the system, any remaining faults can be contained through trustworthy tile-level privilege enforcement (as implemented in *T2H2*). We assume an instance of *T2H2* is located between each tile and its NoC interconnect.

Note that, emulating the spacial isolation of distributed system nodes, we are agnostic about the semantics and interplay of tile-internal and/or core-level components, e.g., MMUs and their virtualization, copy-on-write, memory protection or recovery functionalities.

In the time domain, although manycores might seem the perfect example of a (closely-coupled) synchronous (distributed) system, reality is a bit different, there are several possibilities for instability. For example, excessive resource use raises the temperature and causes thermal managers to throttle the speed of tiles near this hot spot; interfering access patterns reduce memory bandwidth by evicting cache lines from shared caches; and NoC-level bursts may cause noticeable and, with unfair arbitration, unbounded message delays. Faulty behavior (accidental or malicious) might further worsen these negative time-domain effects. A strict synchronous model would not reflect reality and thus be proved brittle.

We rely on a partially-synchronous model and prepare *Midir* for possible delays (notably by buffering consensus votes in *Midir*'s *T2H2*). Two particularities exist in these closely-coupled environments, in contrast to large-scale distributed systems, which play in our favor: (i) barring delay variations, liveness is normally guaranteed; and (ii) the infrastructure is plastic in terms of timeliness trade-offs. Therefore, as in most contemporary BFT approaches, we consider asynchrony for safety and partial synchrony for liveness.

The structure of our protocols is time-free, and as such they remain safe in the presence of delay oscillations, provided that the fault assumptions hold (no more than $f$ tiles get compromised, as discussed below). Then, the protocols inherit whatever synchrony they achieve from the timeliness of the infrastructure they are immersed in: the manycore works with high performance, in execution and communication, exhibiting short and bounded delays during long enough periods of time, but can exhibit significant variations in these bounds. These are fair expectations, considering the nature of these systems.

## 4. Threat model

Our threat model considers software-level compromise at all levels, including in the hypervisor, in the firmware, and, more generally, in any critical software component. This assumption is consistent with our aim of tolerating an incremental level of threat, up to advanced and persistent threats, such as sophisticated attacks mounted by highly skilled and well-equipped adversaries, on tiled manycore systems, often deployed entirely on-chip. Moreover, we consider a limited set of hardware-level faults and attacks: precisely those whose physical effects are confined to a tile (e.g., trapdoors in a core but no hardware faults that cause a chip-wide collapse).

We strive to establish the tile as a unit of component failure. There is no guaranteed fault containment inside tiles. That is, adversaries (or accidents) will be capable of compromising the whole software in any tile (e.g., but not only, a hypervisor replica). Once that happens, we no longer make any assumptions about the correctness of any software in that tile. However, we also consider (and enforce it with the strategy described in Section 3) that tiles themselves are fault containment domains. This and whatever diversification measures are deemed necessary to further support the fault independence assumption for tiles.

We further assume that no more than $f$ tiles are compromised during a reference time $T_a$. Note that this supports the classical fault and intrusion tolerance fault bound for our BFT protocols, but also opens the way to promoting resilience (Sousa et al., 2006). In fact, classical hardening, diversification and intrusion prevention help in putting barriers in the adversaries' way, ensuring that $T_a$ has a usefully large value, and shrinks no further. However, we acknowledge the imperfection of these techniques, especially in face of persistent threats.

We admit that the generic system components (including low-level platform management ones) can be hardened as needed, down to a residual fault and vulnerability rate. As we discussed earlier, this is good, but not enough, especially under malicious threats. We leverage architectural hybridization to amplify the coverage of the assumptions made in this threat model, by allowing differentiated strategies towards the fault rate targets across system components. *T2H2, Midir*'s trusted-trustworthy components, fall under a more restricted fault model, failing only by crashing, much like USIGs in Veronese et al. (2013b). We remind that *T2H2* is hardware-based and executes no software.

We assume it is infeasible to construct and/or verify software or hardware of reasonable dimensions, to a 0-defect goal. However, we stipulate that it is possible to design ultra-reliable, ultimately trusted-trustworthy *simple* components, to a 0-defect target. The consequence is that these will remain correct and operational, despite compromise of the local tile. As discussed in Veríssimo (2006), this is an extremely powerful combination in algorithmic terms: trusted components used routinely to assist critical mechanisms and algorithms (e.g. privilege enforcement, redundancy management) overcoming the residual fault and vulnerability rate of most system components, in order to achieve correct operation with extremely high probability.

This is only possible if we strive for absolute simplicity (for verifiability, e.g. by proof assistants) of these trusted-trustworthy components. That is the case of *T2H2* in the *Midir* hybrid architecture, providing just two generic functions staged, *in hardware*, at the tile-to-NoC interface: access control (capability registers, in-

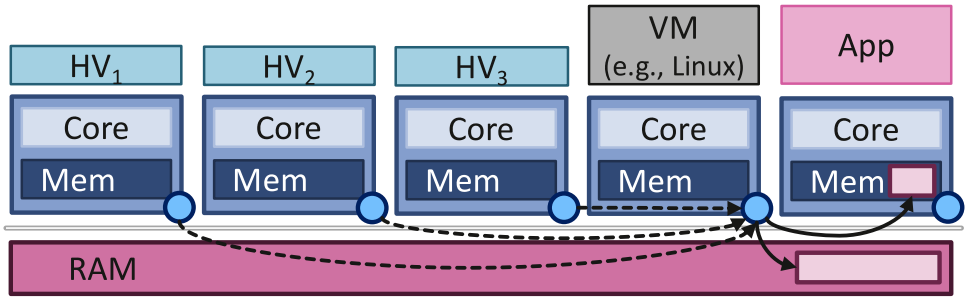


**Fig. 1.** Overview of the *Midir* architecture: a multi-/manycore system augmented with *T2H2* hardware capability units (blue dots) at the NoC interface. Access to tile-external resources is subject to privilege confirmation in *T2H2* and possibly voting. Here, the hypervisor replicas $HV_1, \ldots, HV_3$ consensually reconfigure the privileges of the VM on the 4th core, which in turn obtains access to a region of memory in the scratchpad memory of the application on tile 5. Privilege change is a voted upon operation, indicated by dashed lines. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

cluding the logic for privilege enforcement when tile hard- or software invokes capabilities) and quorum-based consensus (voters). Fig. 1 depicts this layout.

## 5. The *Midir* architecture

As discussed earlier, *Midir* is an architectural concept based on augmenting manycore systems in a minimally intrusive way through strategically placed, simple and self-contained trusted-trustworthy components (*T2H2*). In fact, *T2H2* provides just two generic baseline functions staged *in hardware* at the tile-to-NoC interface: access control (capability registers) and quorum-based consensus (voters).

Fig. 1 depicts one possible layout, of a stereotypical hyper-visor-based system, where the hypervisor is replicated for fault/intrusion tolerance, serving virtualized operating systems and applications: hypervisor replicas are distributed across tiles, so that each replica executes on a different tile, separate from applications; tiles and software therein interface with each other through the NoC; and *T2H2* perform that interconnection.

As long as the execution in a tile remains within the resources associated to this tile (local caches, memories, accelerators, etc.) no overhead occurs, since *T2H2* is not involved in authorizing or denying these accesses. In fact, we remind that it is not the purpose of *Midir* to provide fault containment between software components co-located *on the same tile*. This is like the internal behavior of nodes in a distributed system, where nodes are the unit of fault containment.

Once software components are spread across tiles, they interact through external operations (e.g., via a resource in another tile, via shared on-chip memories or via external memory or IO). In this case, *T2H2* interposes such accesses and validates that each of them has sufficient privileges (i.e., the invoked capability in the *T2H2* capability registers conveys this access). Consequently, hardware faults inside a tile or accidental or malicious faults in any part of the software it executes are limited in propagation to the objects authorized by these capabilities.

Further to capability checking, *Midir* is capable of subjecting these accesses to voting by means of distributed components in different tiles. This is especially important for critical operations, be it in application execution or in platform reconfiguration, in order to achieve some form of fault/intrusion tolerance, from error detection, or self-checking by comparison, to error masking by consensus. To vote, tiles must hold a capability to the corresponding voter, which authorizes this tile to make proposals as one of these distributed components. Voting is mandatory to install new or change existing capabilities, in order to prevent faulty hypervisor replicas from bypassing the aforementioned fault containment when reconfiguring the resources a tile can access.

Given the nature of *Midir*'s trustworthy mechanisms (*T2H2*) is to provide fault isolation, access control and a means of consensus on critical operations, not all write operations, be them at low-level or application-level, make use of *Midir*. As mentioned, in-tile accesses are not interposed by *T2H2* and not every off-tile write needs voting, given not all operations are critical, in the sense of having the potential to modify sensitive memory locations that, if used maliciously, can put the system at risk or place it in a more vulnerable state.

*Midir*'s concept of controlling the tiles' lowest-level privilege enforcement mechanism is agnostic of the mechanism used. However, the simpler such a mechanism and the closer it can be implemented to the tile's NoC interconnect, the more faults *Midir* will be able to tolerate. Hence our choice for capabilities.

Simplicity also governs our voter design. *Midir*'s voters merely collect and act upon proposals of related operations from different components, letting the voted-upon operation proceed. Because tile-external resources are typically memory mapped, these operations are normally simple writes. The voters themselves implement no error handling or diagnostics functionality, but provide information for the voting replicas to perform these tasks. More precisely, voters suspend voting on disagreement, freeze the proposals made and expose them for diagnosis. Moreover, they implement a sequence number $seq_i$ for progress tracking, which they increment after each vote unless the vote gets suspended. A voted upon `voter-reset` operation resumes voting and, as well, increments $seq_i$. Section 7 shows how we utilize this error handling support and Section 8 details our voter implementations.

## 6. *T2H2* – Midir's trusted-trustworthy component

In this Section, we provide further details about *T2H2*.

### 6.1. Voted and non-voted operations

To retain the flexibility of the software in a manycore system, allowing it to dynamically adapt resource-to-application mappings as needed, *T2H2* supports direct access to tile-external resources. This way, applications possessing a capability can directly invoke operations on external resources (e.g., to access read-shared or private data in RAM or to interact with non-critical devices). The scenario in Fig. 2 illustrates a non-voted (write) memory access by Tile A, performed by invoking a capability in this tile's *T2H2* (1). Since *T2H2*'s capability register $c_1$ holds a read-write capability to the memory region $[p, p+s]$ (2), the operation to write value *val* in variable *a* is authorized (3).

However, *T2H2* also supports voting, particularly useful when, e.g., platform management software or hypervisor replicas further have to execute critical operations (e.g., privilege change or criti-

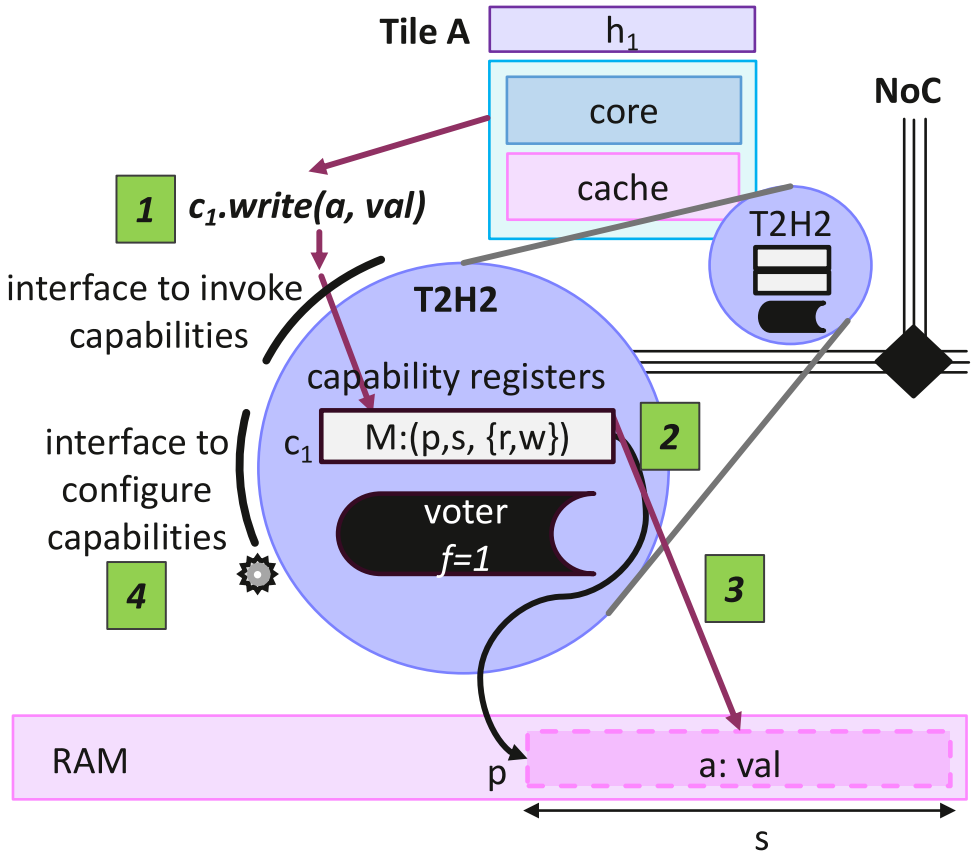


**Fig. 2.** Capability-mediated access of tile-external resources. Invoking capability register $c_1$, application $A$ invokes memory capability $M : (p, s, \{r, w\})$ to write *val* to location $a$ in region $[p, p+s]$. The numbers are guides reflected in the steps explained in the text.

cal device accesses). These operations are voted upon, within preconfigured detection or tolerance mechanisms, to prevent compromised components from causing harm. Several strategies may be served by *Midir*, such as self-checking, recovery blocks, or *f-out-of-n* error masking by majority voting in the presence of $f$ faulty components, but they are all supported by the same baseline voting mechanism. Fig. 3 represents a similar operation as in Fig. 2, but in voted access form. The hypervisor replicas in Tile B and C vote to write value 1, while the one in Tile A, being faulty, votes to write value 0. In order to perform these votes, all tiles invoke a capability on their local *T2H2* to access the designated voter (in this case, the upper voter (orange) residing on Tile A's *T2H2*). Given that a majority of tiles voted to write 1, variable $a$ will be assigned 1.

### 6.2. Consensual privilege change

One particularly relevant scenario for voted access is consensual reconfiguration of the *T2H2* instances themselves. *T2H2*'s reconfiguration interface (see Fig. 2) is accessible only through a voter and cannot ever be invoked directly (4).

Let us understand why this is a relevant innovation. In conventional OS design, any single kernel instance can directly or indirectly enforce modifications on platform resources. So, even in fault tolerant designs, a faulty or compromised kernel instance could still be able to threaten the platform correctness. For example, by manipulating page tables, any low-level OS kernel instance can install virtual-to-physical address mappings to any resource in the platform's memory map and access it through this mapping. Of course, a trusted underlying layer could solve this issue (e.g., by mediating page-table access). However, whether this layer is software, as in the Inktag kernel (Hofmann et al., 2013) or firmware, as in Intel SGX (Costan and Devadas, 2016), it becomes a single point of failure for the platform.

*Midir* provides an additional level of protection, whereby the designer can constrain access to the platform reconfiguration, by allowing a particular mechanism, its registers and data structures to be only effected in a consensual manner, through a voter. As with general voting, discussed in Section 6.1, these voted accesses will normally correspond to the implementation of detection or tolerance strategies, in this case, directed to the protection against threats on the platform itself. In Fig. 3, in green colour (lower voter), we represent such a flow of reconfiguration of a platform capability register in tile A's *T2H2*. Exemplifying with *f-out-of-n* error masking in a replicated low-level kernel, several replicas make the reconfiguration request (dotted line) (1), which is voted (green voter). The result from the voter is wired through a special *T2H2* capability configuration interface to the concerned capability register (2), masking the presence of up to $f$ faulty replicas.

*Midir* does not constrain how systems are configured and hence what faults are tolerated. Instead it provides the means to tolerate an incremental quality of faults, including, for highly critical systems, up to $f$ faults in system management software (e.g., the hypervisor), by providing $n = 2f + 1$ hypervisor replicas and by subjecting all critical operations to voting.

## 7. Towards fault and intrusion tolerant microhypervisors

The *Midir* architecture and, more specifically, the *T2H2* units can be recursively applied at any software layer, protecting any layer of the system's software stack. Both sides of the software spectrum, i.e., low-level management software (e.g., a microhypervisor, mi-

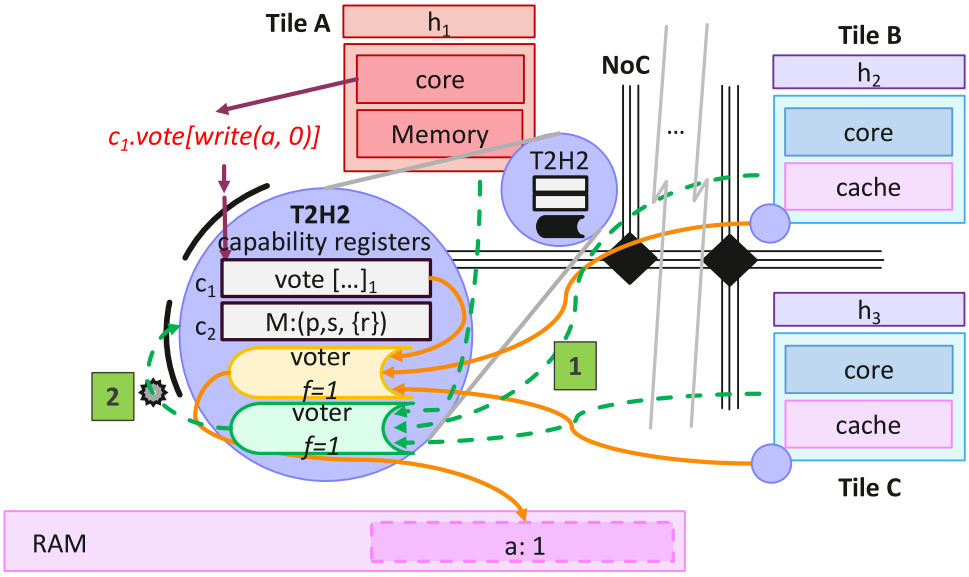


**Fig. 3.** Consensual update of location *a* in the tile-external memory block (upper voter) and consensual reconfiguration of capability register $c_2$ in the *T2H2* of tile *A*. Reconfiguration is always consensual (requiring agreement of a majority of the tiles *A*, *B* and *C*); tile-external resources may be optionally treated in that manner (by granting access to a voter, but no direct access). The voter installs the majority decision (e.g., it updates location *a* with the consensual value 1 or the capability in $c_2$ with the agreed upon read-only memory capability).

crokernel, firmware) and applications, can benefit from the solutions presented in this paper. However, given low-level software is typically a system's last line of defense as a result of the platform management responsibilities attributed to it, and, given the hypervisor's role of performing critical functions pertaining to isolation, access control and privilege enforcement; we believe it illustrates the most complex and costlier usage example of the *Midir* architecture, while providing a concrete solution to the problem described in Section 1. A similar example could be applied to the construction of a fault- and intrusion-tolerant microkernel. At application-level, on the other hand, one can use the *Midir* mechanisms to, for example, coordinate collaboration among several applications sharing data or enforce consensual actuator execution in the context of embedded systems.

We now turn our attention to the construction of *Midir*-aware FIT microhypervisors, such as suggested in Fig. 1. Hypervisor replicas execute on dedicated tiles, from where they remotely configure the privileges of applications executing on other tiles. Most of the other common OS-functionality (e.g., context switching, inter-process communication, (non-critical) device access, etc.) can be left to the application and its kernel-support libraries.

*Midir* gives the designer latitude to use incremental levels of protection for individual operations or sets thereof. On one extreme, configurations may be allowed where all accesses are direct, and thus unprotected by voting (setting up voters for direct pass-through of proposals, i.e., $f = 0$, to reconfigure capabilities).

On the other extreme, the highest level of protection, while retaining the flexibility of a manycore system, eliminates all software-level single points of failure[2] by subjecting all critical operations to voting. We focus on this facet. The replicated microhypervisor offers a system call interface executed by its replicas, entering a service loop and maintaining data structures used to handle system call requests, which they receive from applications, other replicas (e.g., requesting a privilege they lack for executing a system call) or from hardware (e.g., triggered by device interrupts).

[2] Modulo *Midir*'s *T2H2*, which, justified through its simplicity, we assume will not fail.

We provide an informal proof of the protocol's safety and liveness in Appendix Appendix A.

Remembering that the unit of fault containment in *Midir* is the tile (equivalent to a node in a distributed system) the essential requirement for a fault tolerant microhypervisor design is that the replicas behind critical operations are placed in different tiles, such that they communicate by messages, are subject to *T2H2* access control, and converge on the necessary votes as dictated by the algorithm. In order to fully enjoy the baseline functionality provided by *Midir*, a few additional design principles should be followed:

- **P.1** ***Impersonation prevention:*** Correct replicas must deny any operation with a replica identifier that is already in use (*T2H2* voting relies on identifying the individual replicas through their capability; no two replicas should have a capability to the same voter with the same identifier).
- **P.2** ***Bypass prevention*** Correct replicas must deny any operation attempting to grant direct write access to a consen-sual-update-only object (Section 6.2).

Let us illustrate the design with the example of reallocating the tile to a different application. Signaling the tile, an application-specific library may save the state necessary to resume execution (e.g., utilizing memory assigned for this purpose). The actual switch then proceeds by resetting the tile followed by installing the capabilities the new application's library needs, in order to load its state. Obviously, reset and, as we have seen, privilege change are critical operation, which must be performed consensually to prevent compromised kernel replicas from prematurely stopping applications. Channeling such critical operations to voters and confining access with capabilities prevents faulty replicas from causing harm, since, as long as no more than $f$ replicas become compromised, a correct majority out of the $n = 2f + 1$ replicas will outvote these operations. This turns system call execution into updates of replicated state and a sequence of voted operations, which we shall later call *subordinate votes*. This works as well with any other replicated critical software, even firmware such as in SGX (e.g., preventing enclave misconfiguration) or device drivers, when interacting with the physical world. Replies to system calls must also be voted upon, given that hypervisor replicas, by nature, act

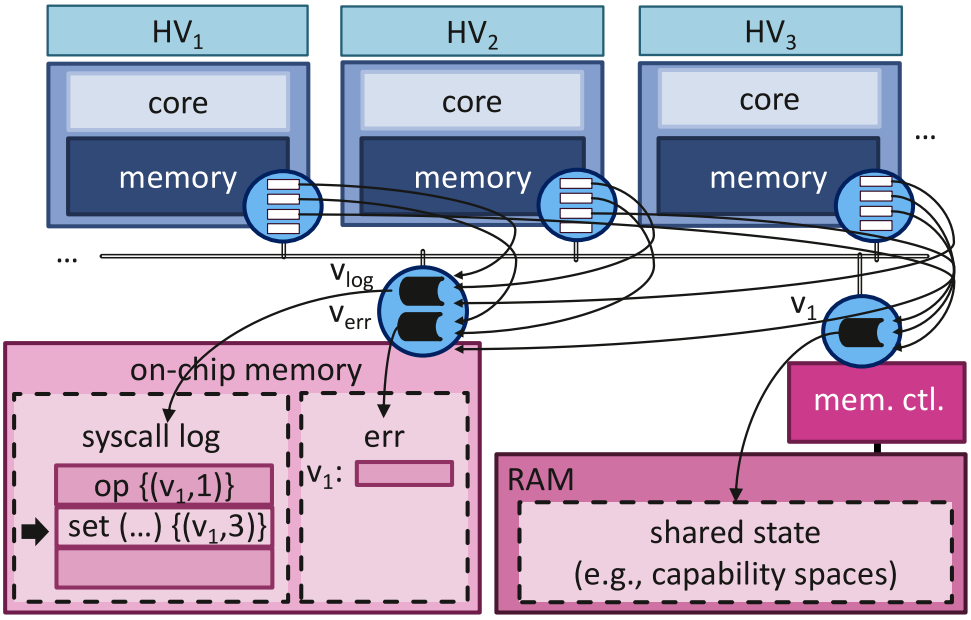


**Fig. 4.** Read-shared, consensually updated data structures used by the kernel: system calls are recorded in the syscall log, the error log keeps voting error information, and a capability space holds an application's capabilities (Section 9).

on behalf of multiple applications, possibly storing information of one that must not be revealed to others.

The above is of course true provided replicas have reached agreement on the system call to execute and on the parameters with which the client has invoked this call. Clients are applications and other kernel replicas that invoke system calls. A further role of the in-kernel service loop is therefore to reach consensus on system call execution order and parameters. From our evaluation (Section 9) we found that *Midir*'s support for consensually executing critical operations also provides for accelerating the BFT protocol that the kernel replicas must execute to reach agreement.

### 7.1. Consensual system calls

Fig. 4 provides a more detailed picture of how *T2H2*'s voters and capability registers contribute to reaching consensus about the system call to execute and its parameters. The service loop of FIT hypervisors needs to reach consensus before it can start executing operations that may have critical side effects when misused.

The service loop utilizes two data structures: a consensually updated ringbuffer — the *syscall log* — records agreed upon system calls and its parameters to give kernel replicas the opportunity to learn about those agreed upon. Otherwise, this information would only be available to the agreeing quorum of $f+1$ replicas and if faulty replicas participate there, but refuse to execute the system call later on, too few correct replicas would have obtained this knowledge to complete the system call. Storing agreed upon system calls in the log allows lagging replicas to catch up with the system calls they missed.

Similarly, the service loop utilizes an *error log* to protect error information from getting lost if the voter is reset prematurely before all replicas have learned about this error. Updates of the syscall and error logs are made through dedicated voters: $v_{log}$ and $v_{err}$, respectively.

Macroscopically, clients place system call requests in authentic buffers, which the kernel replicas poll[3] for new requests. Consensual privilege change allows creating such buffers by granting write access to a single client, but to no kernel replica. The leading kernel replica proposes one such system call by initiating a vote with $v_{log}$, which followers observe and agree or deny. Once written to the syscall log, replicas proceed by executing the system call and the votes for its critical operations, as well as responding to the client. We call these *subordinate votes* as they depend on the main vote, logging the system call. That is, no correct replica will engage in a subordinate vote unless the system call has been logged. Subordinate votes include at least replying to the client and advancing the syscall log to the next free slot. They are performed utilizing a set of voters $V = \{v_1, \ldots\}$ that is disjoint from $\{v_{log}, v_{err}\}$.

We make no assumptions on the order in which replicas update their local state (even transactional or speculative updates are imaginable). However, to simplify tracing the progress of the system call (and, in turn, the code that late or rebooted replicas have to execute to catch up), we require subordinate votes to be executed in the same order by all replicas and assume that this order is completely specified by the system call parameters.

Our rationale for agreeing on the system call first is to circumvent a fundamental problem of consensus protocols without authenticators: the impossibility to diagnose faults if messages can be altered during multicast operations (Lamport et al., 1982). In our setting, cryptographic operations would come at over-proportionally high costs relative to the speed of the transport medium (the NoC). Since consensus adds to system call execution times, having an execution time close to the NoC's speed is a desirable property. We therefore avoid sending unforgeable authentication tokens (e.g., HMACs) and instead exploit the authentication we obtain from a client being the single writer of its request buffer. Additionally, given clients maintain write access to their request buffers, they can change the request after the leader has proposed it, but before followers validate it, which makes it impossible for followers to distinguish whether the leader proposed a wrong system call or whether the leader proposed the client's original suggestion, but the client changed it afterwards. In consequence, they cannot differentiate faulty clients from faulty leaders to provably identify the leader as faulty. We omit this form of error diagnosis for the system call vote to regain this property when we need it:

[3] Sleep/wake protocols can be used in periods where no requests are pending.

```
1   agreement:
2    seq_i := v_i.seq
3    if (replica = seq_i mod n) {
4     // leader
5     v_i.propose(op, seq_i)
6    } else {
7      // follower
8      wait for leader proposal: op
9      validate op
10     if (valid) v_i.confirm(op, seq_i)
11      else       v_i.decline(op, seq_i)
12   }
13   // all
14   wait for f+1 replicas to
15    agree/disagree/timeout
```

**Fig. 5.** Generic voting pattern used in the service loop and when executing system calls.

in the subordinate votes for reaching agreement on critical operations.

Leaders tricked into such a fault are rotated and the new leader proceeds with all other pending requests before returning to the suspicious client.

The following details the protocols the hypervisor replicas execute to reach consensus on and execute system calls. Leveraging the generic voting pattern in Fig. 5, replicas first reach agreement on the system call (Fig. 6) to then consensually perform critical updates during its execution (Fig. 7).

### 7.2. Generic voting pattern

Fig. 5 shows the generic pattern and how replicas interact with voters. Evaluating the sequence number $v_i.seq$ of voter $v_i$, replicas identify the leader as the replica with identifier $v_i.seq \bmod n$[4] in its capability. The leader proposes a request by invoking its vote capability to write operation *op* to its voter buffer, which the voter prevents from being changed once the leader marks this proposal as complete. Followers wait for the leader to complete its proposal to then validate the operation and express their agreement/disagreement (by submitting the operation they saw or by writing the corresponding value to the agreement vector (see Section 8)).

### 7.3. System call vote

In Phase 1, replicas first agree on the system call to execute following the generic pattern above. In Phase 2, they then vote on critical operations. Fig. 6 shows the pseudocode for system call agreement. Lines 16–23 illustrate the client invocation pattern discussed above. The leader selects a pending system call (Line 26) with a valid opcode (Line 27) and prepares the entry to log. To prevent equivocation during subordinate votes (e.g., attempts to trick a replica into proposing the next system call without completing the current one), we enforce some additional principles:

- **P.3** ***Coordinated subordinate votes:*** correct replicas vote only on subordinate voters ($v_i \in V$) to execute the current system call.
- **P.4** ***Presence of correct replica:*** no voted operation succeeds without at least one correct replica.

We enforce P.4 by requiring quorums of at least $f+1$ matching votes, while preventing impersonation (c.f., P.1 in Section 7). In combination, these principles ensure that subordinate voters $v_i \in V$ will keep their state while in Phase 1 (including their sequence numbers). By agreeing, alongside the system call, on the first sequence number of all voters used in this system call (collected in Lines 29–33 in the set *VS* and validated in Line 42), we ensure that all replicas know all sequence numbers to start with in subordinate votes, even if they have been lagging behind. In the absence of errors, the $j$th subordinate vote on $v_i$ will be executed with sequence number $seq_i + j$, assuming $(v_i.seq_i) \in VS$ was the start sequence number of $v_i$. This agreement on the initial sequence number then allows for a simpler progress tracking in Phase 2, when executing subordinate votes.

Because of the impossibility in Section 7.1, system call votes operate with reduced error diagnostics: replicas reset $v_{log}$ if it got suspended after disagreement (Lines 43, 44) and repeat votes for pending system calls unless they fail for all client-leader combinations, in which case they exclude this client.

[4] As long as enough tiles are available, $n$ and $f$ can be reconfigured, namely when adopting optimistic voting schemes. Such changes can namely de done on the go, provided a safe initialization, rejuvenation and relocation protocol. However, we leave the dynamic modification of these parameters and associated advantages for discussion in future work.

```
16 client c_k:
17  write m := syscall opcode + parameters
18   to c_k's request buffer
19  wait for reply in c_k's response buffer
20 hypervisor replica HV_i:
21  service loop:
22   poll all client buffers
23   remember new request (m, c_k) as pending
24   on pending request:
25    // leader
26    (m, c_k) := pending.remove_head
27    if (m is invalid syscall)
28     skip to next pending request
29    VS := ∅
30    for each voter v_i used to execute m
31     // collect voter sequence numbers
32     introspect v_i to read seq_i := v_i.seq
33     VS := VS ∪ {(v_i, seq_i)}
34    // follower
35    if (pending requests ≠ ∅)
36     set timeout
37    // all
38    v_log.agree_on ("write(log, ⟨m, c_k, VS⟩)")
39     with validate :=
40      (m ≠ request from client c_k) ||
41      (v_log.seq ≠ seq_log) ||
42      (seq_v ≠ v.seq, where (v, seq_v) ∈ VS))
43    if (at least one replica disagrees)
44     v_log.vote_for_reset()
45    if (not f+1 agreement)
46     repeat vote
47    execute m
```

**Fig. 6.** Service loop - Phase 1: agree on next system call to execute.

### 7.4. Subordinate votes

The code for executing subordinate votes in Fig. 7 has to solve two problems:

1. preserve determinism despite errors and
2. prevent replicas from prematurely resetting voters.

From reaching agreement on the system call, we know that the first subordinate vote on $v_i$ starts with $seq_i$ because $(v_i, seq_i) \in VS$. As such, without errors, the $j$th subordinate vote on $v_i$ happens with sequence number $seq_i + j$. The same applies to votes with at least one disagreeing replica that all received $f+1$ agreement because, after the voter resets (Line 62), they are not repeated (Line 66). The key for lagging replicas to catch up in case of error is to make sure they learn about all errors, so that they know how many times a vote was repeated and when it was successful. Assume the $k^{th}$ subordinate vote ($k < j$) was the last to fail with $seq_i^k$, then $k$ completed with $seq_i^k + 1$ and the system call progressed to subordinate request $j$ if $v_i.seq - seq_i^k = j - k$.

Solutions to the second problem address the point that all replicas must learn about errors. With $n = 2f + 1$ and $|Q| = f + 1$, up to $n - |Q| = f$ replicas may lag behind while the remaining $|Q|$ progressed to another subordinate request or even to another system call. In particular, faulty replicas may fail a subordinate vote, but

```
48 HV_i.vote (log, v_i, seq_i, req, m, dest) {
49  if (syscall_log.log ≠ log)
50   return success
51  if (v_i.seq ≠ seq_i)
52   if ((err[v_i].log ≠ log) ||
53       (err[v_i].req ≠ req) ||
54       (err[v_i].eseq > seq_i + 1))
55    return success
56   push_error_and_reset_voter
57   if (!err[v_i].success)
58    repeat vote with seq_i + 1
59  // HV_i is up to speed with the others
60  v_i.agree_on(``write(dest, m)'') with seq_i
61    and validate := (m, dest) is valid
62  if (at least one replica disagrees)
63   push_error_and_reset_voter
64   initiate recovery
65  if (f + 1 agreement)
66   return success
67  repeat vote with seq_i + 1
68 }
69 push_error_and_reset_voter:
70  error := introspect(v_i)
71  v_err.agree_on(``write(err[v_i], error)'')
72    with validate :=
73     adjust own error information
74     (proposed error = own error)
75  if (error vote fails)
76   v_err.vote_for_reset(eseq)
77   repeat pushing the error
78  v_i.vote_for_reset(seq_i)
```

**Fig. 7.** System call execution - Phase 2: subordinate votes and error handling.

agree to reset the voter, which erases the error information about the failed vote from the voter and leaves behind as few as a single correct replica to know about the error. This scenario occurs if $f$ faulty and one correct replica resets the voter before others diagnosed it. Clearly, without costly cryptographic information, the honest replica cannot convince others about what has happened. The following design principle solves this problem by preventing premature resets before error information is pushed to the error log.

- **P.5** ***No reset before error logging:*** correct replicas reset subordinate voters only after the error got logged.

This error state contains information about the current system call, i.e.: the system call entry *log*; the subordinate vote *req*; the sequence number of the voter $v_i$; the point where it failed *eseq* and which replicas agreed/disagreed. In consequence, lagging replicas can validate if the current subordinate vote succeeded (Lines 52–55) and, if not, who was responsible for it to fail. Voter $v_i$ prevents destructive writes until it is reset, which P.5 and P.4 ensure happens only after error information was written to the log. Non-destructive writes are updates of empty buffers, respectively, updates of the agreement vector from timeout to agree/disagree and from empty to any of these three.

The argument for why the problem does not recur with the nested vote for logging the error state is as follows:

1. The state to push is held in the voter $v_i$. Therefore, even if a replica lags behind, finding $v_i$ suspended, it knows what information to write to the log.
2. Because of P.5, and because at least $f + 1$ replicas are required (P.4) for votes to succeed, the only way to make progress is by writing correct error information.

Therefore, either faulty replicas agree to writing correct error information or eventually correct replicas catch up and write cor-

rect information. The exact information seen by the replicas may differ depending on the time they read it, i.e., in late reads, more replicas may have expressed their consent or disagreement. However, it will always contain at least the consensual result of the vote, i.e., whether $f+1$ replicas agree, disagree or timed out, and, in the former two error cases, it identifies at least one replica that diverges from the majority (the leader, in case of $f+1$ disagreement). This replica is proven faulty. Followers, reading error information after the leader and finding proposals of additional replicas, downgrade their own information to that of the leader after validating it as described above (Line 73). Repeating the vote while rotating the leader ensures that valid error information is proposed latest after $f$ retries. It then suffices to reset $v_{err}$, whenever it becomes suspended (Line 76). Once error information is pushed, replicas vote to reset the voter $v_i$ for the subordinate vote (Line 78) and continue executing it.

## 8. Implementation

The implementation of capability invocation is standard (Needham and Wilkes, 1974): *T2H2* is invoked by tiles to perform external operations, then it looks up the capability in the capability register file, and forwards the operation to the NoC after the privilege check succeeds, silently dropping the operation otherwise. Replica IDs are communicated as labels in the capability (Hardy, 1985), which *T2H2* inserts as an additional parameter into the operation.

Our voter implementation is driven by the following considerations and their impact on functional simplicity.

### 8.1. Buffered vs. unbuffered votes

Perhaps most impactful is the decision to buffer votes to allow replicas to make their proposals without first having to synchronize on the time when the signal for such a vote must be held. Although buffering increases the complexity of the voter, it decouples replicas, allowing them to act in a partially synchronous fashion and, as long as different voters are used, even partially out-of-order[5] Buffering votes is ideal in a NoC architecture, since votes are transmitted as normal messages (e.g., writes to the memory mapped registers of the voter). Tiles can continue executing once the message is sent. We therefore implement voters to contain buffers for storing proposals from the different replicas for the current vote executed with this voter.

### 8.2. Immediate vs. deferred masking

A similarly impactful decision is whether voters should be able to mask faults immediately. Alternatively, voting can be repeated until a valid proposal is made. The consequences, besides time to agreement, are the amount of memory needed for buffering votes vs. the complexity of the voter logic.

To mask faults and reach agreement immediately after $|Q| = f+1$ matching proposals arrive, the voter needs to buffer suggestions from at least $f+1$ replicas. Since up to $f$ such messages may be wrong and because the voter can only find out after receiving $f+1$ matches, buffer space for at least $f+1$ messages is needed to prevent having to repeat the vote.

We implemented two variants of *T2H2* voters to evaluate the resource/performance trade-off at the two extremes of this spectrum. Our $n$-buffer variant (Fig. 8a) implements one message buffer per replica. Each time a message arrives, it is compared against all other stored messages and the operation applied once $f+1$ buffers match. Our single-buffer variant (Fig. 8b) trades agreement time for a more resource-efficient implementation: there is only one buffer; and only the current leader is granted write access to this buffer. The single-buffer voter follows a leader-follower voting scheme, with the leader proposing a vote and followers validating this proposal. To prevent inconsistency, the voter prevents modification of the leader proposal once the leader marks the proposal as ready. This allows follower replicas to observe the stored message and express their agreement/disagreement. For this purpose, the single-buffer voter implements an agreement vector with one (initially empty: - ) tri-state cell for each replica to express agreement *A* or disagreement *D*. Now, one of three things may happen when replicas propose:

(i) a majority of $f+1$ or more replicas disagree with the leader proposal. In this case, the leader proposal is considered invalid and the operation is not applied; or
(ii) a majority of at least $f+1$ replicas agree. In this case, the proposal is accepted and the voter applies the operation in its buffer.
(iii) the operation times out without a majority of replicas agreeing/disagreeing. In this case, the replicas record this error and repeat the vote after rotating to the next leader.

The $n$-buffer version requires logic circuits for pairwise buffer comparison, whereas in the single-buffer version a 2 data-bit majority gate over the agreement vector suffices, deeming the latter more resource efficient. On the other hand, although the single-buffer voter guarantees that, latest after repeating the vote f times, a healthy replica is elected as leader and makes a valid proposal, the $n$-buffer version may proceed as soon as it finds $f+1$ matching proposal, making it more efficient in terms of execution time.

### 8.3. Internal vs. external error handling

The third question is whether the voter itself should include provisions for diagnosing errors and for informing replicas about them. Errors are detected when one replica diverges with the majority decision. Voter-initiated error handling translates to the voter tracing back to the voting replicas' cores to identify where to deliver error-handling interrupts. The expected complexity discourages such a solution. We therefore offload error handling to software and support replicas by a means to track progress (the sequence number *seq*) and by suspending voting after detecting a mismatch. In this situation, *seq* does not advance but the voter may still apply the operation (in case of $f+1$ agreement). Replicas read the voter registers and buffers to diagnose the error, by looking for divergences.

To resume execution of suspended voters, replicas reset the voter, which clears all buffers and the agreement and reset vectors and advances the sequence number by one. Reset itself is a voted operation over the reset vector, which contains one bit per replica. The voter resets once $f+1$ bits in this vector are set. Although this quorum guarantees that at least one correct replica agrees to resetting the voter, it does not prevent faulty replicas from resetting the voter prematurely, that is, before all correct replicas were able to retrieve the error state. P.5 and the protocol in Section 7.4 handles this corner case.

### 8.4. Dimensioning voters

The last question we discuss here is: for how many faults should the voter hardware be laid out. Since we aim at implementing voters in silicon, we have to make this choice at system design time to dimension buffers and vectors large enough for the

[5] To simplify monitoring of the progress of a system call, we have required that all replicas execute the critical operations of each system call in the same order. Operations of different system calls need not be constrained in this way, and, at the cost of a more complex progress tracking, this requirement can be further relaxed to: same order as far as a single voter is concerned.

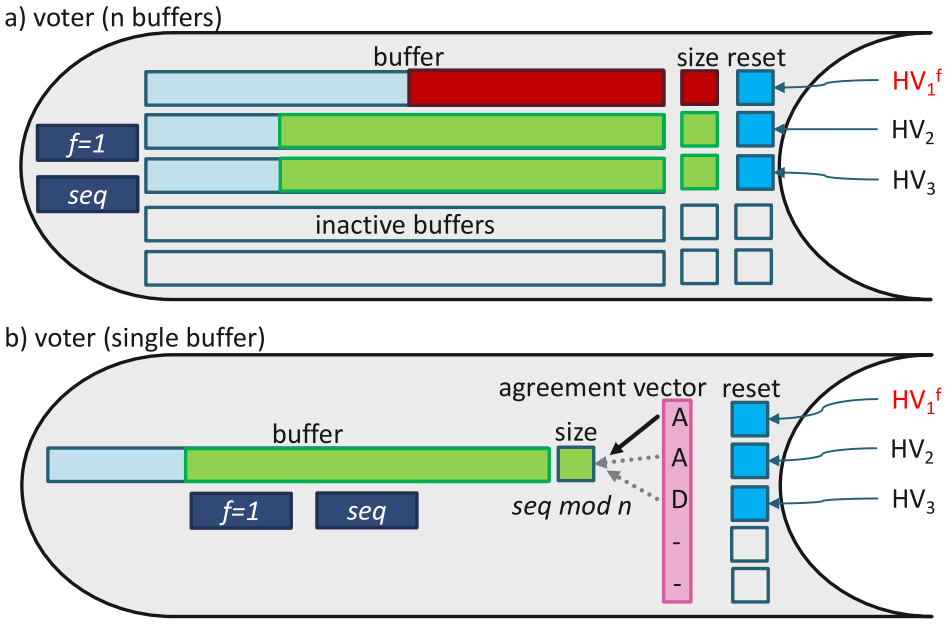


**Fig. 8.** Internal structure of a voter. One, resp. $n$ buffers hold the message of replicas to vote upon and *size* its length. $f$ defines the fault threshold, *seq* is a voter maintained sequence number. The agreement and reset vector are described below.

maximum number of faults to tolerate ($f_{\max}$). However, to not always have to execute at this maximum replication degree, a fault threshold $f \leq f_{\max}$ of voters can be configured at boot time. For instance, if the system should tolerate up to $f_{\max} = 3$ faults, it needs to be dimensioned to have $n_{max} = 2f_{\max} + 1 = 7$ fields in the vectors (and an equal amount of buffers in the $n$-buffer variant). This voter can be operated at any fault threshold $0 \leq f \leq f_{\max}$.

The voter design has been kept simple enough, and decoupled enough from the surrounding logic. As such, we can expect with high confidence that *T2H2* can be implemented and shown correct, as well as stay functional even when the tile it is associated with fails. A crashed *T2H2* prevents its tile from invoking any operation on tile-external resources, in particular from issueing votes. *Midir* ensures safety and liveness as long as the overall number of faulty tiles (including those with a crashed *T2H2*) does not exceed $f$.

## 9. Evaluation

As an early validation of our proposal, we have implemented *T2H2* with both voter variants in VHDL on a Zynq-7 ZC702 Evaluation Board. We instantiated 3 Microblaze cores as tiles, running at 50 MHz, each with one *T2H2*, connecting the tiles through *T2H2* with an AXI interconnect (serving as the NoC). We have implemented and measured the performance of the service loop of a fault- and intrusion- tolerant hypervisor (Fig. 6). The service loop is used to agree on and execute client-invoked system calls for two critical operations: granting and priming capabilities. Grant (`L4.map` (Liedtke, 1995)) copies capabilities between capability spaces and prepares for later revocation. Prime consensually copies a capability from the client's capability space into a *T2H2* capability register, where it is ready for invocation. We have measured the performance of grant and prime in two different implementations of capability spaces, a container object for the capabilities an application possesses:

(i) as a private data structure in each replica (Section 9.1), requiring, in the case of prime, only the vote to install capabilities and two further to reply to the client and mark the system call as finished; and

(ii) as a read-shared, consensually-updated data structure, trading off speed for a smaller memory footprint by introducing additional votes for track keeping (Section 9.2).

As baselines, we compare to a cross-tile invoked singleton kernel (horizontal line), executing the same system calls on its private state, with 1637 cycles for *grant* (1977 cycles for *prime*); and to a shared-memory variant of MinBFT[6] requiring 242824 cycles to agree on a system call. Our agreement protocol outperforms MinBFT by one order of magnitude.

A comparison to a cross-tile invoked singleton kernel allows us to understand the overhead the *T2H2* introduces in remote memory block access, which is present only in the execution of critical operations. The presented values for this baseline are justified by the absence of caches, as we want cores to be as decoupled as possible. The choice for comparison with MinBFT relates to its high efficiency and state-of-the-art popularity in hybrid BFT solutions.

An evaluation of application performance in a *Midir* architecture shall be left for future work.

### 9.1. Per-replica capability space

Fig. 9 shows the average performance of the **grant** and **prime** system calls in a per-replica capability space implementation relative to two baselines: **null** and a singleton kernel instance performing these system calls in a non-consensual manner. Shown are the system calls broken down into individual votes and the Q5 / Q95 percentiles of the overall measurements.

The minimal costs for learning about a system call request and executing it are 1571, 1637 and 1977 cycles on average for null, grant and prime, respectively, which is the baseline of the singleton kernel. System calls for the single-buffer version have a factor of 8.9 (null) to 9.6 (grant) increase, which can be explained due to the voter not benefiting from caching. Whereas the singleton kernel merely has to copy one request from the memory where the client core places it, missing in all caches in the process, following replicas have to poll the voter to wait for the leader to make

[6] We omit client signatures in favor of authentic buffers, but implement UIs with HMACs. USIGs can be accessed without overhead.

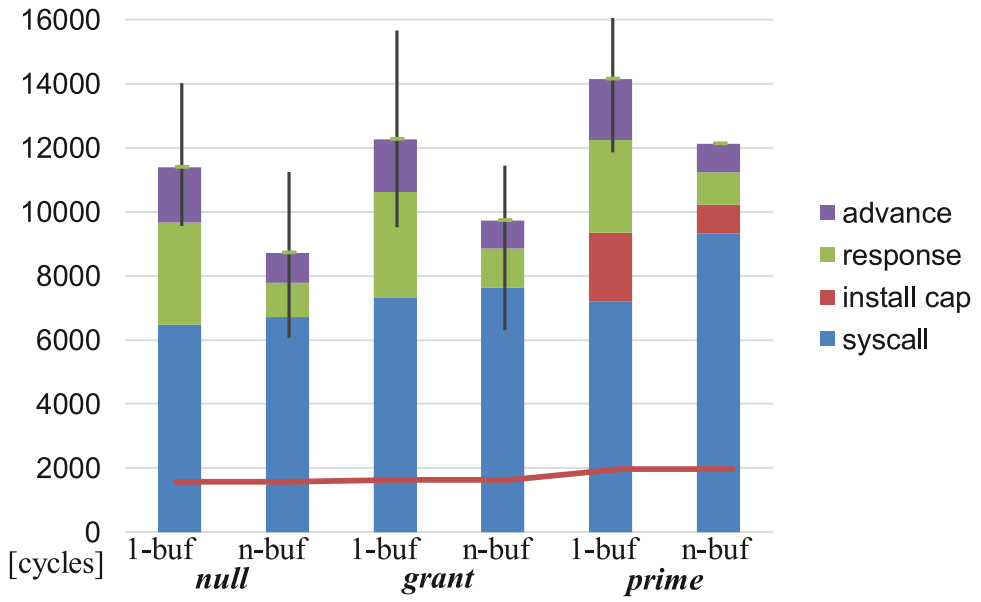


**Fig. 9.** Average execution times of the three consensual system calls — **null, grant** and **prime** — when executed on a per-replica capability space implementation. System calls are broken down into the individual votes for agreeing on the system call and for performing the critical updates required. Shown are also the Q5 / Q95 percentiles and the average costs of executing the respective system calls on a singleton kernel.

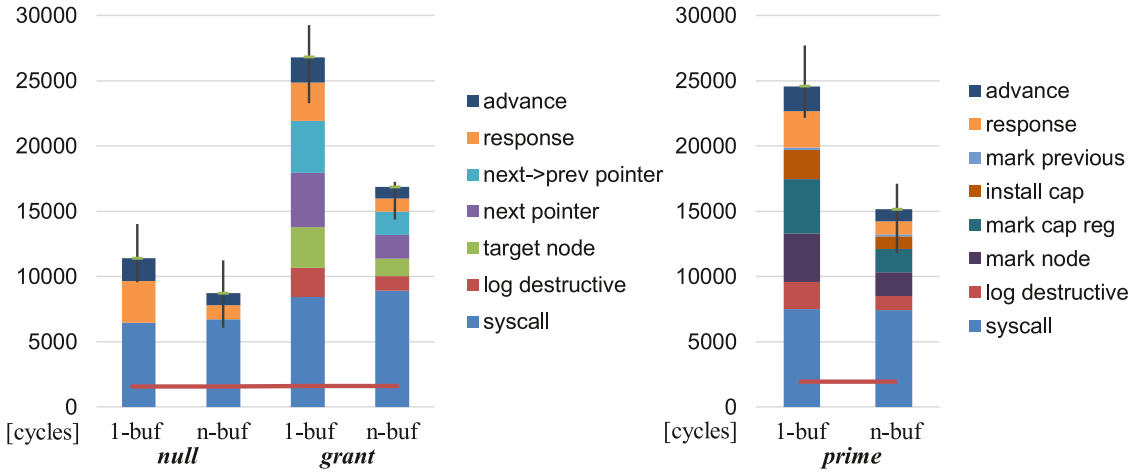


**Fig. 10.** Average execution times of the three system calls for consensually-updated capability spaces.

a proposal and then confirm (or reject) the proposal made. Each such voter access amounts to costs equivalent to a cache miss.

As can be seen, reaching agreement on the subordinate votes is much faster, since replicas already align themselves when reaching agreement on the system call to execute.

In the n-buffer version, higher costs occur during the agreement on the system call, which is due to the writing of the complete request to the voter, not just setting a bit in its agreement vector. However, subordinate votes are much faster, since replicas no longer wait for the leader to make a proposal. Instead, they just propose what should be written as critical operation.

### 9.2. Consensually-updated capability space

Fig. 10 shows a similar diagram as Fig. 9, this time, however, for consensually-updated capability spaces. Granting and priming capabilities now require additional votes to update the data structure.

This time, the 6.7 (single-buffer) and 7.3 (n-buffer) times slower performance relative to the singleton kernel can be explained due to the voter not benefiting from caching:

*Singleton kernel:* System call execution is triggered by the client writing to shared memory on one core and the kernel (on another core) reading it. From then on, all the operations happen locally in the core of the kernel without any interaction with the outside. Therefore, all memory operations aside from the invocation and reply hit in the core's cache, which, in our setting, responds within 1 cycle. The cross-core operations (invocation (1) + reply (2)) dominate these costs.

*Replicated kernel:* System call execution starts as well with invocation (1), but then, the leader needs to propose the request (2), followers validate it (3) and express agreement (4) upon which the voter updates the memory and all replicas wait for the vote to reach agreement (5). In the case of the per-replica capability space (case (i) in Section 9), we then execute locally, but for replying (to not introduce storage channels) we have to repeat at least (4) +

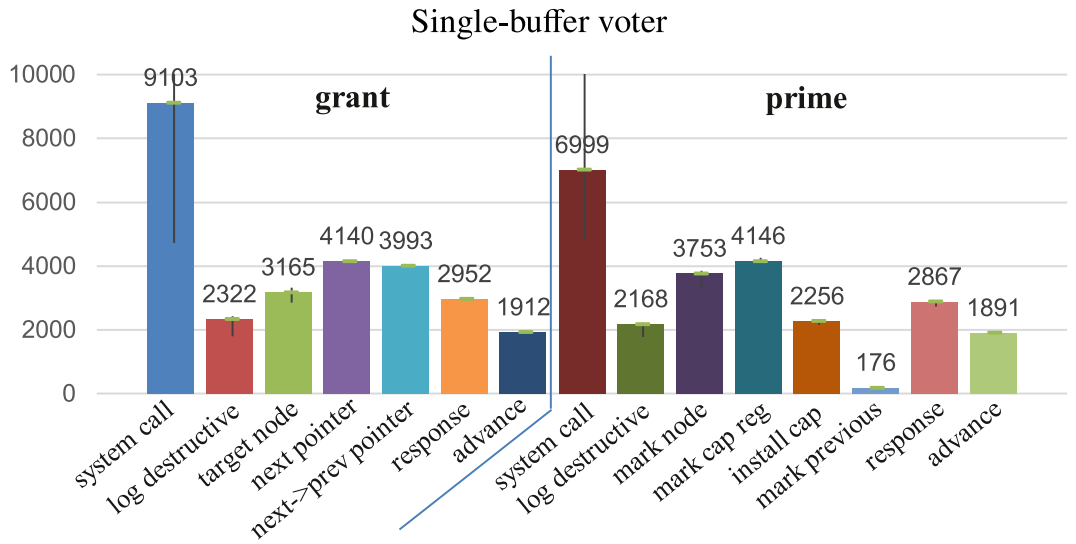


**Fig. 11.** System calls broken down into individual votes. Shown are the Q5 and Q95 percentiles for the main system call vote and each subordinate vote for single-buffer voters.

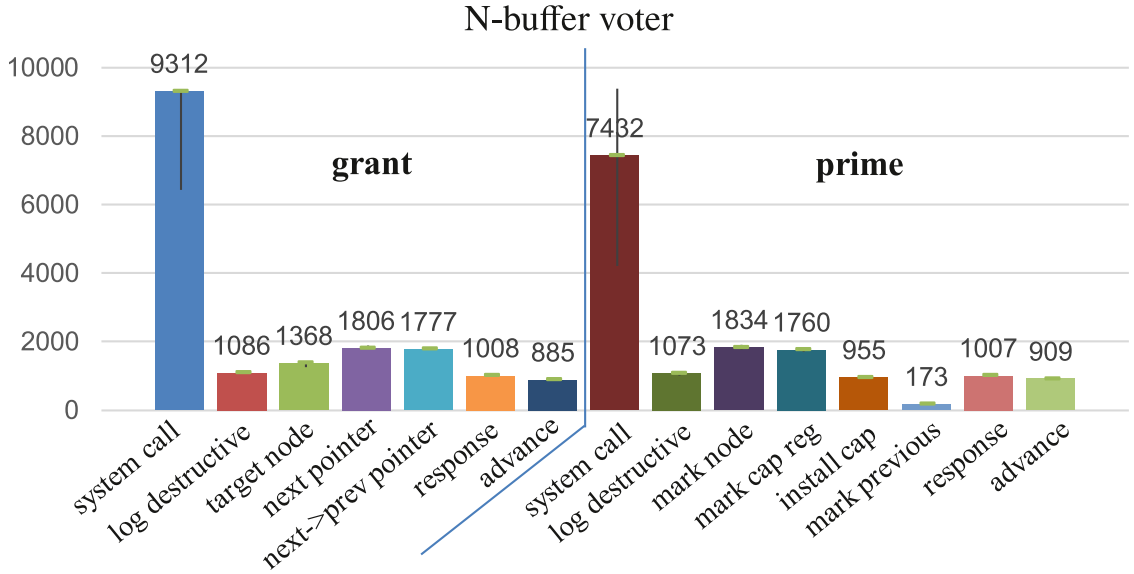


**Fig. 12.** Same as Fig. 11 for n-buffer voters.

(5), assuming *n*-buffer voters. As such, even without any delays, we have 7 cache misses vs. 2 in the singleton kernel execution, hence a factor of 3.5. Additionally, more voter accesses are performed to read the sequence number, which we need for flow control.

To confirm that variations in fact originate from the agreement on the system call to execute, we have broken down system call execution into their individual votes and measured their Q5 and Q95 percentiles. Figs. 11 and 12 show these values for single- respectively n-buffer voters. As expected, subordinate votes remain close to their average execution times, whereas agreement on the system call varies significantly.

### 9.3. Overhead discussion

Given the worst case scenario of an 8,9 (null) to 9,6 (grant) factor overhead of voted accesses in comparison to the singleton kernel (when using per-replica capability spaces), we discuss here the arguments addressing this concern.

First, we remind the reader that not all system calls require voting, with the latter being applied only to execute critical operations (e.g., privilege management) and access critical resources external to the tile. Similarly, if used by software at higher levels of abstraction, namely application level, *T2H2* would as well only be required to perform specific operations that could potentially cause harm, such as those updating critical shared data or accessing critical devices.

Furthermore, critical resources being accessed, provided they present read-most patterns, do not impact overall performance, as reads do not usually require voted access, unless, for security reasons, the information contained therein should not be leaked to specific sets of replicas. On the other hand, if the resource is to be updated (i.e., written), the ratio of reads to writes will determine the impact the aforementioned overhead will have on performance. Also, not all writes (only critical ones) need voting and, thus, not all writes incur the demonstrated overhead. Capability checking incurs 99 cycles overhead to write the request in the input register and 106 to check the permission result. However,

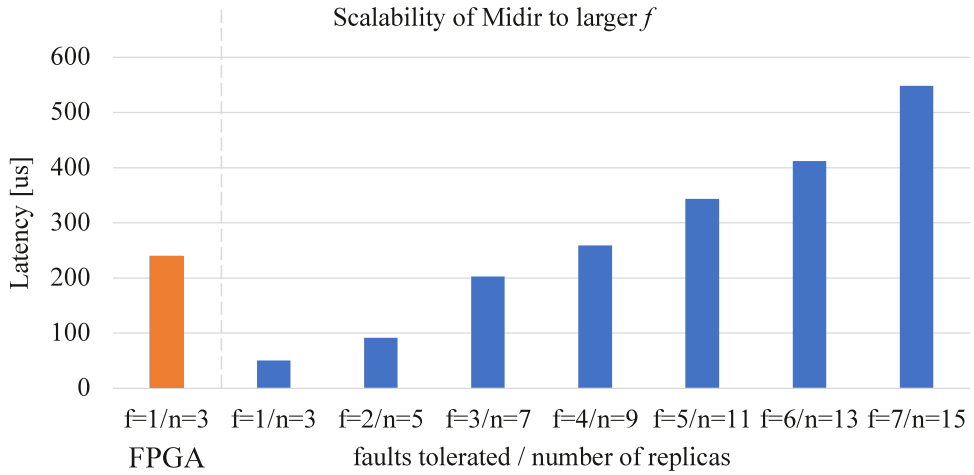


**Fig. 13.** Latency of the `null` system calls for increasing number of replicas in microseconds.

| | Single Buffer | N-Buffer |
|---|---|---|
| Common Definitions | 129 Lines of C++ | |
| T2H2 Interface | 142 LoC++ | 134 LoC++ |
| Service Loop and Subordinate Votes | 311 LoC++ | 309 LoC++ |
| Capability Space (per replica) | 242 LoC++ | |
| Capability Space (consensual) | 314 LoC++ | |
| Capability Registers | 46 / 605 Lines of VHDL | |
| Voter | 187 / 1512 VHDL | 176 / 1703 VHDL |

**Fig. 14.** Code size in lines of C++ / VHDL code (logic / total).

we believe that, although the overhead represents almost an order of magnitude increase, it is still within the expected performance metrics for the considered communication medium - the NoC.

Finally, was this FPGA-based proof of concept built as an ASIC (application-specific integrated circuit), as intended for a final MPSoC product, voted system calls' performance would immediately increase, given an ASIC's die is purposefully built for the task it is designed to perform and, thus, is optimized in terms of area, logic gate count and frequency. Additionally, an MPSoC specifically designed with *Midir* integration would have the *T2H2*s sit much closer to the replica, at its tile-to-NoC interface, with *T2H2*-internal registers that would act as a local cache, thus reducing access times and, combined with an ASIC nature, logic-processing times. In our FPGA proof of concept, the *T2H2*s are instead memory mapped, meaning the processor has to access the external memory block to perform each voting step, thus having the increased overhead explained in Section 9.2.

Nevertheless, this overhead essentially translates to a trade-off, where either no safety measures are applied by removing redundancy and/or voted execution, leaving single points of failure unresolved, but achieving better performance; or having the presented performance decrease while enhancing the system with the safety features proposed in this paper. As mentioned in Section 7, *Midir* may be configured in such a way where all accesses are direct, and thus unprotected by voting ($f = 0$), meaning all *T2H2*s are "turned-off". As such, depending on the system's goals, criticality and requirements, *Midir* can be tailored in regards to not only which operations should be subjected to voting, but the total number of faulty replicas it should tolerate, if any. This, in turn, further adjusts performance as not all available tiles need to be used.

### 9.4. Scalability

Since our FPGA board's resource limitation prevents us from instantiating more replicas, we confirm the scalability of our approach in an emulation on x86. Hypervisor replicas are pinned as the sole application on the cores of a 24-core Intel Xeon CPU E5-4650 system, running at 2.10 GHz. They execute the same server loop like on the FPGA, but emulate voters in software.

Fig. 13 shows the latency results of scaling the null system call to an increasing number of replicas and hence an increasing fault threshold from $f = 1$ to $f = 7$. Also shown (although not directly comparable) is the performance of the FPGA implementation, both scaled to microseconds. As can be seen, the execution of the null system call scales linearly with the number of replicas, which in part is due to the emulation having to acquire a lock during voting. We expect a similar though less steep linear increase in a larger scale FPGA implementation due additive effects of having to wait for the agreement of an increasing number of replicas with fluctuating system-call execution times.

### 9.5. Code size

Fig. 14 lists the code size (excluding initialization) for the service loop, for consensually executing critical operations and for in-

| | Capability Unit (20 cap. regs) | Voter Single Buf ($f_{max} = 1$) | Voter N Buf ($f_{max} = 1$) |
|---|---|---|---|
| Slice LUTs | 750 / 1292 | 2230 / 3532 | 4438 / 5365 |
| Slice Registers | 3367 / 4351 | 3994 / 5983 | 6228 / 7702 |
| F7 Muxes | 115 / 307 | 0 / 290 | 0 / 736 |
| F8 Muxes | 42 / 138 | 0 / 97 | 0 / 352 |

**Fig. 15.** FPGA resources required by *T2H2* (without / with AXI interface).

terfacing with the capability registers. Also shown are the VHDL source lines of code for the logic only and for the overall design (including I/O declaration) of the voter and capability unit. As can be seen, the amount of code that each replica executes for the above grant and prime system call is well below 1000 lines of code. Faults in this code are masked by the majority of replicas outvoting faulty replicas in critical operations. Similarly, the hardware overhead is just above 400 lines of VHDL code for the logic plus 2411 lines of VHDL for connecting the logic to the AXI interface I/O and for mapping the corresponding internal signals. VHDL simply defines the logic to be programmed in the board, it is not executed by the voters or capability units.

Fig. 15 shows the FPGA resources of the (post-synthesis) implementation of our components. LUTs are units with no state, used to implement the combinatorial logic; while registers hold state, e.g, to keep buffer contents, but implement no logic. Each F7 Mux (wide multiplexer) combines the outputs of two LUTs together, while F8 Muxes combine the outputs of two F7.

Notice that the absolute resource requirement of T2H2 will not increase if more complex cores are to be controlled. Hence, the relative resource overhead will shrink when more complex tiles are considered. This phenomenon occurs since the complexity of the cores has no influence on the *T2H2*s' logic and functional requirements. *T2H2* will provide the same services with the exact same hardware logic design independently of the complexity of the tile invoking it. As such, as the cores' resource requirements increase due to higher complexity, *T2H2*'s remains the same. However, resource utilization will increase if more cores are added. Additional input registers will be needed to store an additional request or vote from the new cores as well as LUTs to check these registers when counting votes.

## 10. Conclusions and future work

We have introduced *Midir*, an architectural concept which breaks new ground and opens promising avenues in the applicability and resilience of manycore architectures (MPSoC). Through minimalist mechanisms integrated in the MPSoC architecture, *Midir* frees MPSoCs from the SPoF syndrome, fulfilling the vision of *distributed* systems-on-a-chip (DSoC).

In this paper, we show in particular that *Midir*-enabled DSoCs achieve a quantum step towards off-the-shelf chip resilience, since these mechanisms are generic enough to support, in-chip and with high reliability, a large variety of the protection and redundancy management techniques normally implemented in software at higher layers in 'macro' systems. To convincingly prove our point, we exemplified and evaluated an implementation, over *Midir*, of the most complex version of our solution set: a Byzantine fault tolerant microhypervisor. We have shown the practicality of our concept, having quite satisfying performance, since it outperforms the highly efficient MinBFT protocol by one order of magnitude. The low overhead of our approach shows large promise for future full hardware solutions.

Furthermore, *Midir* was intentionally designed as a non-intrusive extension to current core architectures, being anchored on simple and self-contained hardware extensions, sitting at the tile-to-NoC interface. Taken up by a hardware manufacturer or integrator, it allows a backwards-compatible, non-fracturing evolution, as updating critical resources or re-writing privileges translates to writing specific memory regions, which the capability registers can be configured to point to, at boot time, by the bootloader.

We hope that our findings may be key to enhance general MPSoC architectures towards distributed DSoCs and, among other avenues, lead to next-generation COTS resilient chips.

After this initial work, several questions remain to be answered, namely on kernel design details, rejuvenation and diversification for sustainability, application-level uses, real-time applicability, coverage for network attacks, dynamic reconfiguration of deployed parameters and so forth, which leave ample room for future work. Namely, application-level usage gives way to complex questions. There are two ways *Midir*can be used at application level: a) for applications managing critical resources, for example, cyber-physical controller applications managing a resource that interacts, for instance, with the physical world or b) to construct resilient building blocks used to upgrade mechanisms that coordinate the sharing of critical resources, e.g., more resilient data structures to be shared among several applications within the same system. Future work for a) must explain how the application benefits from *Midir*, which specific operations are voted upon, how real-time requirements are managed and how voting-related error handling impacts the application. For b), an extensive analysis can be done with regards to the several options to create said building blocks, how synchronization among applications is regulated, how this synchronization affects performance and the RCB, how implementation options affect the level of required synchronization (and, thus, performance) and what race conditions can emerge from adapting, e.g., data structure operations.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## CRediT authorship contribution statement

**Inês Pinto Gouveia:** Methodology, Software, Investigation, Writing – original draft. **Marcus Völp:** Conceptualization, Methodology, Software, Investigation, Writing – original draft, Supervision, Funding acquisition. **Paulo Esteves-Verissimo:** Conceptualization, Investigation, Writing – review & editing, Supervision.

## Acknowledgements

Funding: This work was supported by the Fonds National de la Recherche (FNR) [C18/IS/12686210/HyLIT].

## Appendix A. Safety and liveness

In this Section, we argue about the safety and liveness of the BFT protocol for processing system calls (as shown in Figs. 6 and 7). That is, any two healthy replicas execute the same system calls in the same order (safety) and all correct system calls will be eventually executed (liveness). We assume the combination of a sleep-wake notification mechanism and polling (summarized in Line 22) reveals any pending system call to all replicas. However, before we start arguing about safety and liveness, let us see why faulty replicas cannot trick healthy ones into participating in votes with a wrong sequence number.

System call execution involves as set of voters: the subordinate voters $v_i$ mentioned in $VS$, plus $v_{log}$ and $v_{err}$. By construction, voters ignore proposals and confirmations for all sequence numbers other than the current one and only if voting is not suspended. That is, a voter $v_i$ will only react to commands with a sequence number $seq$ if $seq = v_i.seq$. Sequence numbers advance only if $f+1$ replicas agree to a proposal and no replica disagrees, or if $f+1$ replicas agree to reset the voter due to some error case (e.g., one or more replicas disagreeing with the proposal). From property $P.3$ and $P.5$ and the arguments we have given in Section 7.4 we know that no healthy replica participates in reset before error information has been confirmed by such a replica and logged through $v_{err}$. Moreover, we know that healthy replicas will engage with subordinate voters only for executing the current system call they process. This means either the replica is participating in the current system call or it was lagging behind other replicas. In the latter case, the sequence numbers it will use to invoke the voter are smaller [7] and the voter will ignore the request without any effect.

### A1. Safety

Proposing $VS$ as part of the system call (Line 38) and including this as part of the agreement (Line 42) means a fault-threshold exceeding quorum of replicas agrees to the starting point of subordinate votes and from there we know, from the arguments given in Section 7.4, that without errors the $j$th subordinate vote on a voter $v_i$ is executed at $seq_i + j$ where $(v_i, seq_i) \in VS$ (and similarly with errors, by recording and acknowledging the number of retries). Therefore, if a healthy replica votes for a subordinate vote, it will always vote with the correct sequence number, which implies faulty replicas cannot leverage this vote/agreement to confirm a different request.

From the above, we can conclude safety holds, by seeing that replicas will not agree on different system calls for the same sequence number. The voter will only write system calls to the log which received $f+1$ agreement, and the log position is advanced consensually and in a way that allows all replicas to learn about updates (last subordinate vote of the previous system call). The voter itself thereby prevents equivocation by freezing the proposal the leader makes for the current sequence number, i.e., by preventing it from being overwritten for the current sequence number. Additionally, sequence numbers will not be reused for the same vote since both successful requests and reset advances this number.

From safety of the logged system call, its parameters and $VS$, it then follows safe execution of the subordinate votes, given that the $j$th subordinate vote on voter $v_i$ is completely defined by these aspects. Notice that all healthy replicas execute the logged system calls, including their subordinate votes. In particular, Line 50 will not lead to skipping the execution of the remaining system call, but only short cuts through the subordinate votes when realizing that the system call has already been completed. Healthy, but lagging replicas therefore first update their state with logged system calls before engaging in new system call requests. Notice also that, while it is possible for faulty clients to trick leaders in proposing a system call that followers will not confirm, the consequence of this is merely a rotation of the leader (by reset of $v_{log}$ in Line 43) and the next leader continuing with another client.

### A2. Liveness

What remains to be seen is why the system is live (i.e., why it will eventually process all requests from correct clients). The combination of sleep-wake and polling in Line 22 will iterate through all client/replica combinations. Therefore, each valid client will repeatedly find a correct leader who proposes the request. Partial synchrony then ensures that during the long enough periods of synchronous behavior, healthy replicas engage in processing this request. Let us therefore, for the following argument, assume request processing happens in such a good phase and will not time out. Then latest after rotating through $f$ leaders, the client will find a healthy leader to propose the request.

As shown in Line 14 and 15, replicas will wait for either $f+1$ replicas to agree, $f+1$ replicas to disagree or $f+1$ replicas to time out. Thus, if the request is proposed by a healthy leader (or by a faulty, but stealthy leader in a correct manner) at most $f$ (respectively $f-1$) replicas can disagree and, in the absence of timeouts, $f+1$ agreement will be reached. Then, even if the vote is suspended due to a disagreeing replica, the voter will record the system call in the log and all healthy replicas will proceed by executing the logged call (after resetting $v_{log}$ in Line 44 to return this voter into a state where it accepts further votes, including the next system call).

For subordinate votes, a similar argument applies. In the absence of timeouts during long enough phases of synchrony, when a replica proposes an operation for a subordinate vote, replicas wait until either $f+1$ replicas agree to the proposal (in which case the voter executes the operation, e.g., by writing to the specified destination), even if a minority of replicas disagree; or $f+1$ replicas disagree. Disagreeing replicas causes an error to be recorded and the vote to be repeated. From the arguments in Section 7.4 we know that error logging makes progress latest when a healthy replica proposes a valid error record and when lagging healthy replicas catch up to find the error information in the voter (remember $P5$ prevents premature reset before the correct information is logged). As such, latest after rotating through $f$ faulty leaders a healthy leader will propose and reach $f+1$ agreement (from healthy followers or from stealthy faulty replicas responding correctly). This ensures that each subordinate vote gets executed and, consequently, the system call as a whole. Having seen that the proposed BFT protocol for system call execution is in fact safe and life, we now focus on the implementation of the voters and how it ensures the behavior we require, namely freezing proposals and suspension until consensual reset.

[7] We assume sequence numbers used by lagging replicas will never be overtaken by the current write and say that such a sequence number is smaller, despite possible wraparounds of the used integer. We substantiate this assumption by implementing a large enough sequence number space.

## References

Aggarwal, N., Ranganathan, P., Jouppi, N.P., Smith, J.E., 2007. Configurable isolation: building high availability systems with commodity multi-core processors. In: International Symposium on Computer Architecture (ISCA), pp. 470–481.

Aguilera, M.K., Ben-David, N., Guerraoui, R., Marathe, V., Zablotchi, I., 2019. The impact of RDMA on agreement. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. Association for Computing Machinery, New York, NY, USA, p. 409418. doi:10.1145/3293611.3331601.

Aguilera, M.K., Ben-David, N., Guerraoui, R., Marathe, V.J., Xygkis, A., Zablotchi, I., 2020. Microsecond consensus for microsecond applications. 14th USENIX Symposium on Operating Systems Design and Implementation.

Al-Boghdady, A., Wassif, K., El-Ramly, M., 2021. The presence, trends, and causes of security vulnerabilities in operating systems of IoT's low-end devices. Sensors 21 (7), 2329.

Asmussen, N., Völp, M., Nöthen, B., Härtig, H., Fettweis, G., 2016. M3: a hardware/operating-system co-design to tame heterogeneous manycores. Architectural Support for Programming Languages and Operating Systems. ACM, Atlanta, GA, USA.

Avizienis A., Chen L., et al. On the implementation of n-version programming for software fault-tolerance during program execution1977;.

Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A., 2009. The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09. ACM, New York, NY, USA, pp. 29–44. doi:10.1145/1629575.1629579.

Bhat, K., Vogt, D., van der Kouwe, E., Gras, B., Sambuc, L., Tanenbaum, A.S., Bos, H., Giuffrida, C., 2016. Osiris: efficient and consistent recovery of compartmentalized operating systems. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 25–36. doi:10.1109/DSN.2016.12.

Biggs, S., Lee, D., Heiser, G., 2018. The jury is in: monolithic OS design is flawed. Asia-Pacific Workshop on Systems (APSys). ACM SIGOPS, Korea doi:10.1145/3265723.3265733.

Bolchini, C., Carminati, M., Miele, A., 2013. Self-adaptive fault tolerance in multi-/many-core systems. J. Electron. Test 29 (2), 159–175. doi:10.1007/s10836-013-5367-y.

Bressoud, T.C., Schneider, F.B., 1995. Hypervisor-based fault tolerance. In: 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado, USA, pp. 1–11.

Brooks, T.T., Caicedo, C., Park, J.S., 2012. Security vulnerability analysis in virtualized computing environments. Int. J. Intell. Comput. Res. 3 (1/2), 277–291.

Castro, M., Liskov, B., 1999. Practical byzantine fault tolerance. 3rd Symposium on Operating Systems Design and Implementation. ACM, New Orleans, USA.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., Gupta, A., 1995. Hive: fault containment for shared-memory multiprocessors. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95. ACM, New York, NY, USA, pp. 12–25. doi:10.1145/224056.224059.

Colman-Meixner, C., Develder, C., Tornatore, M., Mukherjee, B., 2016. A survey on resiliency techniques in cloud computing infrastructures and applications. IEEE Commun. Surv. Tutor. 18 (3), 2244–2281. doi:10.1109/COMST.2016.2531104.

Correia, M., Neves, N.F., Verissimo, P., 2004. How to tolerate half less one byzantine nodes in practical distributed systems. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, pp. 174–183. doi:10.1109/RELDIS.2004.1353018.

Costan, V., Devadas, S., 2016. Intel SGX Explained. Technical Report. Massachusetts Institute of Technology. https://eprint.iacr.org/2016/086.pdf(Accessed: 2016-07-22)

Das D.. An indian nuclear power plant suffered a cyberattack. Here's what you need to know. https://www.washingtonpost.com/politics/2019/11/04/an-indian-nuclear-power-plant-suffered-cyberattack-heres-what-you-need-know/; 2019. Accessed: 2017-03-12.

David, F.M., Chan, E.M., Carlyle, J.C., Campbell, R.H., 2008. CuriOS: improving reliability through operating system structure. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08. USENIX Association, Berkeley, CA, USA, pp. 59–72. http://dl.acm.org/citation.cfm?id=1855741.1855746.

Davies A.. Tesla's autopilot has had its first deadly crash. https://www.wired.com/2016/06/teslas-autopilot-first-deadly-crash/; 2016. Accessed: 2017-03-12.

Depoutovitch, A., Stumm, M., 2010. Otherworld: giving applications a chance to survive OS kernel crashes. In: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10. ACM, New York, NY, USA, pp. 181–194. doi:10.1145/1755913.1755933.

Döbel, B., 2014. Operating System Support for Redundant Multithreading. Technische Universität Dresden, Dresden, Germany Ph.D. thesis.

Elphinstone, K., Shen, Y., 2013. Increasing the trustworthiness of commodity hardware through software. In: 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).

Engel, M., Döbel, B., 2012. The reliable computing base: a paradigm for software-based reliability. Workshop on SOBRES.

Ermolov, M., Goryachy, M., 2017. How to hack a turned-off computer — or running unsigned code in intel management engine. Black hat Europe, London, UK. Avail at https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf, accessed 15.04.2018

Esposito, E.G., Coelho, P., Pedone, F., 2018. Kernel paxos. 37th Symposium on Reliable Distributed Systems (SRDS). IEEE.

Függer, M., Schmid, U., 2012. Reconciling fault-tolerant distributed computing and systems-on-chip. Distrib. Comput. 24 (6), 323–355.

Gens, D., 2018. OS-Level Attacks and Defenses: From Software to Hardware-Based Exploits. Technische Universität Darmstadt Ph.D. thesis.

Govil, K., Teodosiu, D., Huang, Y., Rosenblum, M., 1999. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99. ACM, New York, NY, USA, pp. 154–169. doi:10.1145/319151.319162.

Greenberg A.. Hackers remotely kill a jeep on the highway. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/; 2015.

Hardy, N., 1985. KeyKOS architecture. SIGOPS Oper. Syst. Rev. 19 (4), 8–25. doi:10.1145/858336.858337.

Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S., 2006. Construction of a highly dependable operating system. In: Proceedings of the Sixth European Dependable Computing Conference, EDCC '06. IEEE Computer Society, Washington, DC, USA, pp. 3–12. doi:10.1109/EDCC.2006.7.

Hoffmann, M., Dietrich, C., Lohmann, D., 2013. Failure by design: influence of the RTOS interface on memory fault resilience. In: G. S. of Informatics (Ed.), Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13). http://www4.cs.fau.de/Publications/2013/hoffmann_13_sobres.pdf.

Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E., 2013. Inktag: secure applications on an untrusted operating system. SIGPLAN Not. 48 (4), 265–278. doi:10.1145/2499368.2451146.

Joseph, M.K., Avizienis, A., 1988. A fault tolerance approach to computer viruses. In: IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 52–58.

Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S.V., Schröder-Preikschat, W., Stengel, K., 2012. CheapBFT: Resource-efficient byzantine fault tolerance. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12. ACM, New York, NY, USA, pp. 295–308. doi:10.1145/2168836.2168866.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S., 2009. seL4: Formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (Eds.), Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11–14, 2009. ACM, pp. 207–220. doi:10.1145/1629575.1629596.

Knight, J.C., Leveson, N.G., 1986. An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Softw. Eng. SE-12 (1), 96–109.

Kocher, P., Genkin, D., Gruss, D., Haar, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2018. Spectre Attacks: Exploiting Speculative Execution. Technical Report. ArXiv e-prints 1801.01203

Kopetz, H., Bauer, G., 2003. The time-triggered architecture. Proc. IEEE 91 (1), 112–126.

Kuvaiskii, D., Faqueh, R., Bhatotia, P., Felber, P., Fetzer, C., 2016. Haft: hardware-assisted fault tolerance. In: 11th European Conference on Computer Systems (EuroSys), London, UK, pp. 1–17.

Lackorzynski A., Warg A., Hohmuth M., Härtig H.. L4re. https://l4re.org/doc/index.html; 2018.

Lamport, L., 1998. The part-time parliament. Trans. Comput. Syst. 16 (2), 133–169.

Lamport, L., Shostak, R.E., Pease, M.C., 1982. The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4 (3), 382–401. doi:10.1145/357172.357176.

Lee D.. Myfitnesspal breach affects millions of under armour users. bbc.com; 2018.

Lee R.M., Assante M.J., Conway T.. Analysis of the cyber attack on the ukrainian power grid. 2016. https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf.

Lenharth, A., Adve, V.S., King, S.T., 2009. Recovery domains: an organizing principle for recoverable operating systems. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV. ACM, New York, NY, USA, pp. 49–60. doi:10.1145/1508244.1508251.

Levin, D., Douceur, J.R., Lorch, J.R., Moscibroda, T., 2009. TrInc: small trusted hardware for large distributed systems. In: Proceedings of the Sixth USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, Boston, Massachusetts, USA, April 22–24, 2009, Boston, Massachusetts, USA, vol. 9, pp. 1–14.

Liedtke, J., 1995. On micro-kernel construction. In: Jones, M.B. (Ed.), Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3–6, 1995. ACM, pp. 237–250. doi:10.1145/224056.224075.

Lipp, M., Schwart, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., 2018. Meltdown (CVE-2017-5754). Technical Report. ArXiv e-prints 1801.01207

Mancini, L., 1986. Modular redundancy in a message passing system. IEEE Trans. Softw. Eng. (1) 79–86.

Matias, R., Prince, M., Borges, L., Sousa, C., Henrique, L., 2014. An empirical exploratory study on operating system reliability. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14. ACM, New York, NY, USA, pp. 1523–1528. doi:10.1145/2554850.2555021.

McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A., 2010. Trustvisor: efficient TCB reduction and attestation. In: 2010 IEEE Symposium on Security and Privacy, pp. 143–158. doi:10.1109/SP.2010.17.

Meserve J.. Mouse click could plunge city into darkness, experts say. http://edition.cnn.com/2007/US/09/27/power.at.risk/index.html; 2007. Accessed: 2017-03-12.

Mullender, S. (Ed.), 1993. Distributed Systems, second ed.. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Needham, R.M., Wilkes, M.V., 1974. Domains of protection and the management of processes. Comput. J. 17 (2), 117–120.

Nikolaev, R., Back, G., 2013. VirtuOS: an operating system with kernel virtualization. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems

Principles, SOSP '13. ACM, New York, NY, USA, pp. 116–132. doi:10.1145/2517349.2522719.
Ogg, S., Al-Hashimi, B., Yakovlev, A., 2008. Asynchronous transient resilient links for NoC. In: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '08. ACM, New York, NY, USA, pp. 209–214. doi:10.1145/1450135.1450182.
Ostrand, T.J., Weyuker, E.J., 2002. The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02. ACM, New York, NY, USA, pp. 55–64. doi:10.1145/566172.566181.
Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2004. Where the bugs are. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04. ACM, New York, NY, USA, pp. 86–96. doi:10.1145/1007512.1007524.
Palix, N., Thomas, G., Saha, S., Calvès, C., Muller, G., Lawall, J., 2014. Faults in linux 2.6. ACM Trans. Comput. Syst. 32 (2), 4:1–4:40. doi:10.1145/2619090.
Patterson D., Ganapathi A.. Crash data collection: a windows case study. 3D Digital Imaging and Modeling, International Conference on2005;280–285. 10.1109/DSN.2005.32
Powell, D., Bonn, G., Seaton, D.T., Verissimo, P., Waeselynck, F., 1988. The delta-4 approach to dependability in open distributed computing systems. In: 18th IEEE International Symposium on Fault-Tolerant Computing (FTCS), pp. 246–251.
Prabahar, B.P., Edwin, B.E., 2012. Survey on virtual machine security. Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET) 1 (8), 115–121.
Price R.. Facebook says it 'unintentionally uploaded' 1.5 million people's email contacts without their consent. Businessinsider.com; 2019.
Recently reported xen/critix hypervisor vulnerabilities, documented in CVE-2019-18420, CVE-2019-18421, CVE-2019-18424, CVE-2019-18425. 2019.
Schiper, N., Rahli, V., Van Renesse, R., Bickford, M., Constable, R.L., 2014. Developing correctly replicated databases using formal tools. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, pp. 395–406.
Schmid U., Steininger A.. Decentralised fault-tolerant clock pulse generation in VLSI chips. 2010. TU Wien, patent: US7791394B2.
Seshadri, A., Luk, M., Qu, N., Perrig, A., 2007. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07. ACM, New York, NY, USA, pp. 335–350. doi:10.1145/1294261.1294294.
Shapiro, J.S., Hardy, N., 2002. Eros: a principle-driven operating system from the ground up. IEEE Softw. 19 (1), 26–33. doi:10.1109/52.976938.
Sousa, P., Neves, N.F., Verissimo, P., 2006. Proactive resilience through architectural hybridization. In: Proceedings of the 2006 ACM Symposium on Applied Computing. ACM, pp. 686–690.
Sundararaman, S., Subramanian, S., Rajimwale, A., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Swift, M.M., 2010. Membrane: operating system support for restartable file systems. Trans. Storage 6 (3), 11:1–11:30. doi:10.1145/1837915.1837919.
Swift, M.M., Annamalai, M., Bershad, B.N., Levy, H.M., 2006. Recovering device drivers. ACM Trans. Comput. Syst. 24 (4), 333–360. doi:10.1145/1189256.1189257.
Szefer, J., Keller, E., Lee, R.B., Rexford, J., 2011a. Eliminating the hypervisor attack surface for a more secure cloud. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 401–412.
Szefer, J., Keller, E., Lee, R.B., Rexford, J., 2011b. Eliminating the hypervisor attack surface for a more secure cloud. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11. ACM, New York, NY, USA, pp. 401–412. doi:10.1145/2046707.2046754.
Tanenbaum, A.S., Kaashoek, M.F., 1994. The amoeba microkernel. In: Distributed Open Systems, pp. 11–30.
Thongthua, A., Ngamsuriyaroj, S., 2016. Assessment of hypervisor vulnerabilities. In: 2016 International Conference on Cloud Computing Research and Innovations (ICCCRI). IEEE, pp. 71–77.
Traverse, P., Lacaze, I., Souyris, J., 2004. Airbus fly-by-wire: a total approach to dependability. In: Building the Information Society. Springer, pp. 191–212.
Tsidulko J.. The 10 biggest cloud outages of 2018. https://www.crn.com/slide-shows/cloud/the-10-biggest-cloud-outages-of-2018; 2018.
Turnbull, L., Shropshire, J., 2013. Breakpoints: an analysis of potential hypervisor attack vectors. In: 2013 Proceedings of IEEE Southeastcon. IEEE, pp. 1–6.
Verissimo, P., Neves, N., Cachin, C., Poritz, J., Powell, D., Deswarte, Y., Stroud, R., Welch, I., 2006. Intrusion-tolerant middleware - the road to automatic security. Secur. Privacy, IEEE 4, 54–62. doi:10.1109/MSP.2006.95.
Veríssimo, P.E., 2006. Travelling through wormholes: a new look at distributed systems models. SIGACT News 37 (1), 66–81.
Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P., 2013a. Efficient byzantine fault-tolerance. IEEE Trans. Comput. 62 (1), 16–30. doi:10.1109/TC.2011.221.
Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P., 2013b. Efficient byzantine fault tolerance. IEEE Trans. Comput. 62 (1), 16–30. doi:10.1109/TC.2011.221.
Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, Anant, 1997. Baring it all to software: raw machines. IEEE Comput. 30, 86–93.
Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M., 2014. The CHERI capability model: Revisiting RISC in an age of risk. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA '14. IEEE Press, Piscataway, NJ, USA, pp. 457–468.
Yang, P., Wang, Q., Li, W., Yu, Z., Ye, H., 2016. A fault tolerance NoC topology and adaptive routing algorithm. In: 2016 13th International Conference on Embedded Software and Systems (ICESS), pp. 42–47. doi:10.1109/ICESS.2016.20.
Yeh, Y.C., 1998. Triple-triple redundant 777 primary flight computer. In: 1996 IEEE Aerospace Applications Conference. Proceedings, vol. 1. IEEE, pp. 293–307.
Yusof N.. Personal data of 808,000 blood donors compromised for nine weeks; HSA lodges police report. TODAYonline; 2019.
Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., Brewer, E., 2006. Safedrive: safe and recoverable extensions using language-based techniques. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. USENIX Association; OSDI '06, Berkeley, CA, USA, p. 4. http://dl.acm.org/citation.cfm?id=1267308.1267312.

**Inês Pinto Gouveia** received her Ph.D in Computer Science from the University of Luxembourg, in 2022. Previously, she completed her Bachelor's and Master's degrees at the Faculty of Sciences, University of Lisbon. Her research interests are in low-level programming languages, namely hardware description languages, systems architecture, distributed systems and fault and intrusion tolerance. She is currently a postdoctoral researcher at SnT, University of Luxembourg.

**Prof. Dr.-Ing. Marcus Völp** heads the Critical and Extreme Computing Group (CritiX) of the Interdisciplinary Center for Security, Reliability and Trust at University of Luxembourg. He received his Ph.D. in 2011 from Technische Universität Dresden, has been visiting scholar at Carnegie Mellon University and was appointed Associate Professor in 2020. His research interests include methods, tools and system architectures for constructing resilient cyberphysical and embedded systems, from small scale to large scale distributed systems. The goal is to simultaneously tolerating accidental and intentionally malicious faults (i.e., targeted attacks), while continuing to guarantee realtime, secure and dependable behavior.

**Paulo Esteves-Veríssimo** is a professor at KAUST University (KSA) and Director of its Resilient Computing and Cybersecurity Center (https://rc3.kaust.edu.sa/), and research fellow of SnT at the Univ. of Luxembourg (UNILU). He is past Chair of IFIP WG 10.4 on Dependable Comp. and F/T. He is Fellow of IEEE and of ACM, and associate editor of the IEEE TETC journal, author of over 200 peer-refereed publications and co-author of 5 books. He is currently interested in resilient computing, in areas like: SDNbased infrastructures; autonomous vehicles; distributed control systems; digital health and genomics; or blockchain and cryptocurrencies.