IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2023.0322000

On-chip replication – Look what the others did

INÊS PINTO GOUVEIA[1], RAFAL GRACZYK[2], MARCUS VÖLP[2](Member, IEEE), PAULO ESTEVES-VERISSIMO[1] (Member, IEEE)
[1]RC3 Center, CEMSE Division, King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia (e-mail: author@kaust.edu.sa)
[2]Interdisciplinary Center for Security, Reliability and Trust, University of Luxembourg, Esch-sur-Alzette, Luxembourg (e-mail: author@uni.lu)

Corresponding author: Inês Pinto Gouveia (e-mail: ines.pintogouveia@kaust.edu.sa).

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

ABSTRACT As resilience challenges evolve, namely in safety- and security-critical environments, the demand for cost-efficient, automated and unattended fault and intrusion tolerance (FIT) grows. However, current on-chip solutions typically target only accidental faults and rely on some form of application-specific redundancy, a single-point-of-failure (SPoF) management software layer or synchrony-reliant protocols. Plus, they are often performance heavy and costly for the emerging tightly-coupled systems in terms of area and power consumption. In this paper, we investigate novel ways to apply high-performance FIT by using replication of a lightweight agreement protocol, *iBFT*, executed with the aid of hardware trusted-trustworthy memory tag accelerators, to avoid misuse of critical operations and SPoFs. We introduce an FPGA-based implementation of *iBFT* under two fault models, evaluate their performance, area usage, and power consumption on a Zynq ZC702 FPGA and compare it with other state-of-the-art protocols. Additionally, we implement and evaluate a software-based emulation of a potential microcode implementation.

INDEX TERMS Fault and intrusion tolerance, hardware, resilience, systems architecture.

I. INTRODUCTION

Many application scenarios, including some that are distributed in nature, cannot tolerate individual nodes failing, or worse, falling into the hands of adversaries. Consider for example a swarm of drones, autonomously-driving vehicles or satellite constellations. Full compromise and hence adversarial control of an individual system already grants hackers and cyber-terrorists the ability to mount cyber-kinetic attacks and the means to cause damage to the environment in which they act. A mission may still be accomplished despite the failure of one unit, but the very same failure can lead, e.g., the affected drone to clash into the others or any other element of the surrounding area. In other environments, humans operate in close proximity to such (cyber-physical) systems, which turns safety into a key requirement, in particular during security incidents.

In many of the above scenarios, it is often not enough to stop the digital control system of compromised individuals. For example, the built up inertia of a vehicle often already suffices to serve the adversaries' purpose, namely to cause damage. In such settings, each individual system must be able to tolerate intrusions [1] and *fail operationally* to constrain adversaries in their attempt to gain full control, until it can finally be recovered to a state at least as secure as its initial one or stopped safely, even if under a degraded mode of operation.

Existing on-chip fault and intrusion tolerance (FIT) solutions protecting individual systems or system nodes, tend to be application- [2]–[4] or OS-specific [5]–[8] and target only accidental faults; rely on a low-level management software layer or hardware components that become a single point of failure [9]–[13]; be based on synchrony-bound protocols [14]; have a considerably large reliable computing base (RCB) [15]; or have high complexity [16]–[22], leading to a non-negligible statistical fault footprint [23]. Additionally, traditional replicated/redundant FIT designs, common in the realm of distributed systems, tend to be costly not only financially, but also in regards to performance and SWaP (space, weight and power) metrics if implemented on multiprocessor systems-on-chip (MPSoCs) or any other sort of tightly-coupled environment.

In this work, we investigate a novel mechanism for constructing highly-efficient on-chip FIT solutions out of independently failing cores, chiplets or sockets integrated in a single board computer such as an MPSoC [24]–[26]. We highlight the design constraints for independent failure, introduce an FPGA-based tagged-memory accelerator for creating write-once memory as an abstraction, and argue why such memories are trusted-trustworthy components. In addition, we discuss how tagged memory can help prevent a common issue in traditional FIT consensus protocols — equivocation, i.e., the possibility of displaying inconsistent information at different points in time to distinct readers — without resorting to using costly cryptographic operations. Finally, we discuss and implement both a hardware FPGA-based version of our solution and a software-based emulation of a possible microcode implementation.

Our solution, albeit inspired by the classical FIT protocols from distributed systems, takes advantage of the tight-coupling of cores to provide high-performance and to minimize the safety/performance trade-off. Additionally, it is designed to not rely on any software layer that can potentially become a single point of failure. Instead, we aim at reducing the reliable computing base (RCB) [27] as much as possible, while balancing it with performance goals. We demonstrate the use of write-once (*wo*) tagged memories in a novel agreement protocol, called *iBFT*, and illustrate how a system can safely execute critical operations (e.g., privilege escalation, access to critical memory regions or I/O devices) that use low-level software layers such as a microkernel or hypervisor.

In this solution, low-level management software, which, if compromised can grant an attacker access over the whole platform and potentially infect all other parts of the system (e.g.,), is replicated across different available cores and each replica votes, in read-shared *wo* memory, whether to execute an operation, similarly to the idea behind dual- and triple-modular redundancy. In particular, our work is heavily inspired by [28], presenting different trade-offs in terms of RCB size, simplicity, overhead, voting, cacheability and memory restrictions.

A. COMPROMISE OF LOW-LEVEL SOFTWARE

Numerous vulnerabilities have been reported in real-time operating systems' (RTOSs) source code, namely in IoT devices (e.g., CWE-119, CWE-120, CWE-126, CWE-134, CWE-398, CWE-561, CWE-563) [29]. Vulnerability analysis of virtualized environments and hypervisor security have shown the various ways these can be attacked [1], [30]–[33], with works such as [34], [35] discussing privilege escalation attacks in hypervisors for full compromise.

Such evidence deems a low-cost and easily verifiable solution necessary.

II. CHALLENGES

Redundancy is often useful to build resilience against benign or arbitrary faults [36]–[39]. It can come in the form of DMR, TMR, or generally in configurations where $n >= f + 1$ replicas detect and $n >= 2f + 1$ replicas mask the behavior of faulty replicas which, in the case of cyber attacks, can be arbitrary, i.e., Byzantine. Redundancy can also come in the form of validating executions at a fine-granularity, e.g., by executing programs in lock-step or TMR and comparing the results of every instruction; or by comparing progress at a coarser scale with the increased benefit that replicas can diverge in between comparison points [40], which improves fault independence. The redundancy and performance costs, however, need to match the intended platform, in this case, on-chip platforms. This means cryptographic operations, traditionally used for ensuring transferable authentication in FIT, become prohibitively high in terms of performance metrics. Furthermore, power consumption should be close to the cost of handling no replication.

Considering the on-chip environment, communication between replicas presents itself as a crucial point in keeping performance costs low. The performance of shared-memory operations (246 cycles for 256 byte and 2331 cycles for 4096 byte transfers, measured with x86's `rep; movsq rep; cmpsq` instructions on an AMD Ryzen 7 3700X 8-Core CPU, 2 threads per core, running at 2.2GHz) as well as their suitability for tightly-coupled systems, encourages consensus to be performed by means of shared memory instead of some form of message passing, like IPC.

Furthermore, in order to obtain performance as optimal as possible, reaping benefit of the tight coupling of replicas, one must minimize reads and writes, meaning replicas should be able to just read a memory region whenever they desire, without having to request that information and wait for it to arrive.

A final consideration is that of equivocation, i.e., the possibility of changing shared memory contents and, thus, presenting different contents at different points in time in the protocol, leading replicas to read different information. This is a problem orthogonal to that of authentication and impersonation. To deal with such issues we use write-once tagged memory to prevent replicas from changing their consensus decisions, which we explain in Section V. For the sake of flow and clarity, we shall explain the details of equivocation later in Section XIII-E.

III. CONCEPT

iBFT implements fault tolerance through light-weight consensus on *critical* operations executed by low-level software (e.g., privilege escalation, access to critical memory regions, handling of CPS I/O device), requiring $n = 2f + 1$ replicas to tolerate up to f faults of an arbitrary kind without requiring a trusted kernel. The fault threshold f is application-dependent and can be decided by the designer/developer. As other FIT algorithms, *iBFT*'s aim is to reach agreement on the order of client operations to execute. In the context of *iBFT*, clients are replicas of low-level software, e.g., an hypervisor, wanting to execute a critical operation that, if performed single handed by a malicious replica, could lead an attacker to gain control over the platform. In essence, *iBFT* is an accelerated form of on-chip consensus that takes advantage of the low overhead of operations like `memcpy` and `memcmp` to achieve an efficient form of fault and intrusion tolerance.

In a summed up manner, when a critical operation needs to be executed it triggers a system call. However, instead of being immediately executed, the low-level software instead must write the *request* for execution as a proposal. Replicas must achieve agreement on whether to execute the operation, based on a majority decision. Since some can be compromised and, thus, faulty, a lightweight consensus protocol,

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

iBFT, must handle the agreement to guarantee only benign requests are executed.

Due to performance goals, instead of message exchange for communicating votes and agreement progress, replicas shall leverage shared local memories hardened with write-once tags (further discussed in Section V) for communicating with other replicas.

Fig. 1 gives a general overview of an *iBFT*-supporting architecture. Shown are the abstract containment domains (tiles), including a core and a (shared) write-once (*wo*) tagged memory whose write ports are exclusively connected to this core. Other cores should be connected through the Network-on-Chip (NoC), or other adopted bus system, only to the read ports of this memory so that they cannot modify their contents. Since each replica receives restricted write access ($rw*$) to its *wo* memory (`t-mem` in Fig. 1) and read-only access to the *wo* memories of other replicas[1], each buffer can be written by exactly one replica. Thus, we have implicit writer authentication, although this authentication is not transferable.

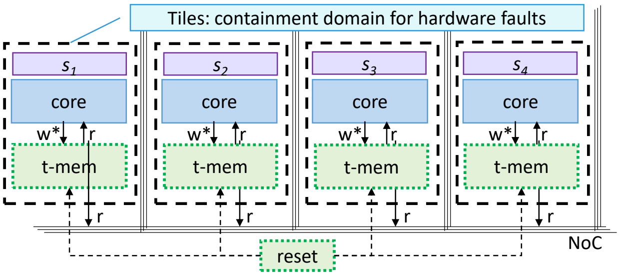


FIGURE 1. ***iBFT* architecture overview.**

IV. SYSTEM AND THREAT MODEL

Before diving into *iBFT* protocol details and write-once tagged memories let us further elaborate on *iBFT*'s system and fault model, as well as its synchrony-related properties.

A. SYSTEM MODEL

iBFT is built for scenarios where a number of tightly-coupled nodes operate in consensus, i.e., vote to reach agreement on a critical operation to execute. We consider hosted as well as bare-metal implementations, e.g., with replicas executing in a single chip on the cores of a multi- or many-core system. In the remainder of this paper, we shall primarily refer to a bare-metal execution of *iBFT*. For bare-metal configurations, we consider tightly-coupled systems to be comprised of sufficiently many cores (as is often the case with MPSoCs) to execute all n replicas $\mathbb{N} = \{s_0, \ldots, s_{n-1}\}$ concurrently, such that $n = 2f + 1$. We follow a model with architectural hybridization [41], where trusted-trustworthy components and other parts of the RCB [27] follow a distinct fault model.

B. SYNCHRONY

Tightly-coupled systems naturally tend to higher degrees of synchrony, but are also susceptible to time-domain attacks, e.g., overheating cores to throttle neighboring ones or denial-of-service attacks in the network-on-chip [42], which makes perfect synchrony assumptions brittle. We therefore assume only partial synchrony [43], i.e., bounded execution and transmission times during 'good' periods, which we assume occur frequently and last long enough to make progress.

C. FAULT MODEL

We tolerate up to f arbitrary faults at hardware or software-level, as long as the physical effects of faults remain confined to the core or the data it produces, including bitflips in local state, wrong computations, among others.

Cores may fail arbitrarily, even at hardware level, but not in a way where such a hardware failure brings down other cores, e.g., no power glitches that bring down neighboring cores and also no faults in the power distribution and clock networks, which are often shared and span large areas of the chip. Of course, conventional multi- and manycore designs retain the possibility of common mode failures in central hardware components like the clock or power distribution network, which must be addressed differently. Resilient clocks [44] mitigate some of these common-mode faults and the recent trend towards interconnected chiplets further improves the physical decoupling of tiles and, therefore, the possibilities for fault containment. Plus, core diversity has become easier with (1) the use of FPGAs, which can create soft cores using off-the-shelf IPs from different vendors or from open-source implementations like RISC-V, (2) dynamic reconfiguration of FPGA partitions through, e.g., Xilinx's Dynamic Function Exchange (DFX), and (3) the emergence of chiplets and their possibility of assembling diverse IPs in a single platform.

Implementations of the trusted-trustworthy component, write-once tagged memory (*wo* for short), may follow distinct fault models of which we consider two flavors, orthogonal to the question of which parts of the hardware to trust:

- Write-once memory implementations that do not fail.
- Write-once memory implementations that can fail, but only by crashing and in a detectable manner.

For the former, we assume these memories to eventually complete read and write operations and to report the last value written. Moreover, they prevent overwriting values that have been tagged (explanation in Section V). In this setting, no further progress guarantees can be conveyed once a write-once memory crashes. Our second trust model considers such crashes. We aim to continue guaranteeing progress unless more than a total of f replicas become faulty or their memories crash. We further assume these memories crash only in a detectable manner. As long as memory value errors build up slowly, the combination of ECCs, memory scrubbing, IPs like Xilinx's Soft Error Mitigation (SEM) core and deliberate

---

[1] Write access is restricted in the sense of allowing values to be written exactly once in between resets

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

crashing[2] (once ECC detects more errors than can be corrected) ensures safety despite crashes. Also notice that we only bound the total number of faults, not distinguishing replicas with a crashed write-once memory from compromised replicas. This aspect will become important for the safety of our approach, since with c write-once memories crashed, we will assume that the remaining system has to cope only with up to $f - c$ compromised replicas. One added benefit of this fault model is that intrusion detection systems may deliberately crash a write-once memory to silence a suspected faulty replica.

V. WRITE-ONCE MEMORY

To deal with the possibility of replicas changing their shared memory contents at will, we need a means to prevent overwriting protocol information once a decision has been made. As such, we introduce write-once (*wo*) tagged memory, a trusted-trustworthy memory abstraction, which leaves reads unconstrained, but prevents successfully written values from being overwritten until the location holding this value is reset (see Section IX). Reset, being a critical operation itself, equally requires voting and agreement from a majority of replicas.

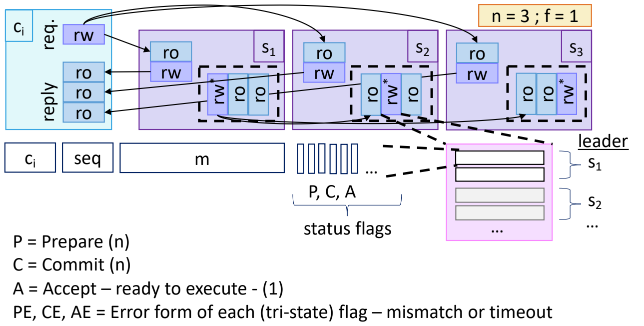
The concept of tagged memory, first introduced in [45] stores values as unions of data and type, making it dependent on the type which operations can be executed on the data. Similarly, we shall use two types of data for *wo* tagged memory:

- *Write-once tri-state bitfields - tags*, whose bits can be set, but not cleared until reset. Bits are split into agreement and error bits, forming together the tri-state. Setting an agreement bit, prevents the corresponding error bit to be set and vice versa.
- *Fixed-size character strings - requests*, for requests, which cannot be overwritten once the string is marked 'ready' (e.g., by setting a bit in a corresponding write-once bitfield).

VI. IBFT PROTOCOL

In *iBFT*, a leader replica encodes client requests in a character string, stores it in its *wo* memory buffer and marks it as 'ready'. Reading this buffer and observing this status, peers detect this proposal and know from its status that the proposing replica can no longer change what is suggested, which prevents equivocation. Therefore, because follower replicas read the same location as the leader, the leader cannot lie inconsistently about the client or its request. Note, it is still possible for a leader to make up a request. Followers express their agreement/disagreement in a similar manner by setting the corresponding bits in a write-once bitfield, which prevents equivocation during this protocol step as well (i.e., a replica indicating agreement toward one of its peers and disagreement to others).

Fig. 2 shows the basic setup of shared memory buffers between the server replicas and local clients. Each client c_i has a request buffer (req) mapped writable to its address space and read-only to the address space of all other replicas. Conversely, service replicas (s_1, s_2, s_3 for $f = 1$ and $n = 3$) use per-client writable reply buffers, which are mapped read-only into the client address space.



P = Prepare (n)
C = Commit (n)
A = Accept – ready to execute - (1)
PE, CE, AE = Error form of each (tri-state) flag – mismatch or timeout

FIGURE 2. Setup and permissions of shared and memory buffers and internal structure of the protocol buffers in *wo* memory.

The *wo* memories are organized in slots. Each slot is comprised of one *wo* character string, used by the leading replica to record the client request m to execute, a client sequence number seq and the identifier c_i of the client, and of n *wo* bitfields (tri-state status flags) for each replica used to store status information and to express agreement. As shown in Fig. 2 there are n prepare (P), n commit (C) and one accept (A) flags, which can also take the form of PE, CE and AE, respectively, to indicate errors (e.g., mismatch). The tagged-memory device ensures that the *wo* string is sensitive to the *wo* bitfield of the leader and ensures that no further modification of the string are possible once a bit is set in the bitfield.

P flags denote a resemblance to the prepare phase in PBFT [46], MinBFT [47] and other BFT protocols and serve the purpose of making sure replicas compared the leader's proposal with the client request. C flags correspond to the commit phase and ensure at least $f + 1$ replicas prepared. A is set to mark the request ready to execute, i.e., after seeing $f + 1$ C flags. The error form of each flag (PE, CE and AE) denote a mismatch or timeout in each phase of the protocol and trigger error handling, being then also used to skip requests. AE is the final tri-state value of A and ensures the A flag is no longer modifiable in case of error, preventing faulty replicas from tricking others into executing requests.

In *iBFT* only the software replicas running the protocol (on different cores) and the write-once memories are replicated[3], as redundancy of other components is not mandated by the protocol. However, communication between the cores and all write-once memories and reset devices is needed, but, since NoCs are now a common means of having all-to-all

---

[2] The main reason a write-once memory would crash itself is when its correction ability for memory faults is exhausted. Deliberate crashing is an additional mechanism, which requires consensus among replicas and is applied only after a replica revealed itself as Byzantine, which cannot be known initially.

[3] Replication of the reset device is also possible and, in fact, recommended.

```
1  client $c_i$:
2      $c_i.req.m$ := $m$
3      $c_i.req.seq$ := $c_i.req.seq + 1$
4      wait for $f+1$ matching replies in $c_i.reply[k]$
5                  from different replicas $s_k$
```

FIGURE 3. Client Code

communication between cores and certain peripherals like memories, this is not an issue.

Let us then describe the behaviour of all involved parties in each phase of the protocol.

A. CLIENTS

Clients c_i store requests in their request buffer (Line 1 in Figure 3) and coordinate with the server replicas by setting the client sequence number[4] $c_i.req.seq$ to a value larger than the previously processed requests. After executing the request, the replicas s_i will reply with this sequence number to indicate that they have completed this request. In particular, this ensures that servers will not confuse requests that remain in the client's request buffer as new, since these requests will have a client sequence number $c_i.req.seq$ that is smaller than or equal to the client sequence number of requests that the server has already processed (Line 11 in Figure 4).

B. NORMAL PHASE

iBFT draws inspiration from [48] and implements a rotating leader scheme, while recording proposals and agreement status in *wo* memory. We start by discussing the *iBFT* pseudo code for error-free cases (shown in Figure 4), before we consider error handling and the code in Figure 5. We have marked in both figures the introspection operations `poll`[5], `copy` and `compare` in green. Lines marked with '*' are required only to cope with crashing *wo* memories.

Replicas take turns as leaders for a configurable number of `slots_per_leader` (Line 9). As long as unused slots are available, leaders insert pending client requests[6] from $c_i.req$ in the next free slot x they control[7], copying the message m, the client sequence number seq and the client number c_i into their buffer $buf_l[x]$ (Lines 10–16) and marking it as complete by setting their P flag (Line 17), which in turn instructs *wo* memory to prevent further writes to this character string.

Followers maintain a timeout for pending client requests to avoid indefinite waiting for a faulty leader not proposing pending requests[8]. To find out when the leader has proposed, they poll the P flag of the leader s_l in the leader's buffer (i.e., $buf_l[x].P[l]$), possibly using sleep/wake techniques to limit contention and to reduce energy consumption (Line 22).

Finding $P[l]$ set, followers know that the proposed request can no longer be changed by the leader. They therefore copy the leader proposal to their buffer (Line 25) and compare it against the proposal made by the client (Line 27). Upon match, they indicate their agreement, by setting the leader's P flag $P[l]$ in their buffer (i.e., $buf_k[x].P[l]$) (Line 28), otherwise, in case of mismatch (or timeout), they set this flag as PE (remember flags are tri-state).

Lacking transferable authentication, replicas cannot distinguish whether (1) the leader is faulty and made up a request, (2) the client is faulty and tricked the leader into proposing a wrong message[9], or (3) both client and leader are faulty. Leaders therefore copy the request into their *wo* memory and followers copy the leader request into their *wo* memories to prepare for the case when the *wo* memory of the leader might crash. Followers s_i compare the leader proposal against the client request and confirm this by setting $P[i]$

After that, leader and followers alike wait for $f+1$ replicas s_j to set their P flag $P[j]$ (Lines 32–36), after which they set their C flag (Line 37) (or CE in case of timeout) and wait until $f+1$ replicas have done the same before they consider the request as ready to execute, by setting the A flag (Line 43). In particular, they confirm before setting P-flags that remote copies match their copy as received from the leader.

Waiting for $f+1$ C-flags set in $f+1$ replicas ensures for the case when $c \leq f$ *wo* memories crash that $f-1$ replicas confirmed the copies in the $f-c+1$ remaining *wo* memories of replicas that participated in this operation. This third round is not required when no further guarantees are given upon *wo* memory crash.

Ready requests are executed by the code (Lines 45–48) once previous slots are executed (or skipped as a result of error handling). Replicas reply by writing both the response and the client sequence number to the reply buffer, which is mapped read-only to the client (Lines 50–51). The consensual reply resets the client buffer[10].

First marking slots by comparing proposals and by setting P flags accordingly, but then delaying execution until all previous slots are executed or skipped, allows for some out-of-order processing without sacrificing linearizability.

We shall return in Section VIII, to checkpoints and the reset operation required to clear the buffer when wrapping around and discuss now how *iBFT* handles errors.

---

[4] We shall use standard C notation for accessing arrays and structures, but allow whole structure copy and compare. For example, $buf_l[x].P[l]$ in Line 17 in Figure 4 refers to slot x in the buffer of replica l, accessing the P flag array in the message data structure at position l. That is, we set the P flag of replica s_l in this replica's buffer at the current request slot x.

[5] The operation `poll` refers to repeated polling until the target is found.

[6] *iBFT* supports multiple clients. The leader, when searching for new client requests, polls different clients, for instance in a round-robin fashion.

[7] In Figure 4, *buffer_length* refers to the number of slots and not the size of the slot.

[8] In a bare metal implementation, both the leader and its followers have no other means than polling to learn about new requests, cycling through all clients in the process. Naturally, this can be quite inefficient as the number of local clients grows. For this reason, we recommend complementing sleep-/wait techniques with some way of informing about the source, triggering the wake up. Hosted setups provide this source information with the replica-invoking inter-process communication.

[9] The word "wrong" here relates to equivocation, i.e., making other replicas believe the leader is in the wrong when, in fact, the client changed the request.

[10] Multiple buffers can be used for each client to amortize reset costs.

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

```
6      server replica $s_k$:
7        /* round 1 */
8        /* next free slot: $x$ */
9        let $l = x$ div slots_per_leader mod $n$
10       if ($s_k = s_l$) /* leader */
11         if $x$ < buffer_length
12           search new client requests
13           on new request $req$ from $c_i$:
14             $buf_l[x].req.c := c_i$
15             $buf_l[x].req.seq := c_i.req.seq$
16             $buf_l[x].req.m := c_i.req.m$
17             set $buf_l[x].P[l]$
18             x := x + 1
19       else /* follower */
20         on new client requests (e.g., $req$ from $c_j$):
21           set timeout ($c_j$)
22         poll $buf_l[x].P[l]$
23         on $buf_l[x].P[l]$ is set:
24           /* found proposal from leader */
25   *       copy $buf_l[x]$ to $buf_k[x]$
26           let $c_i = buf_k[x].req.c_i$
27           if compare $buf_l[x].req = c_i.req$
28             set $buf_k[x].P[l]$
29             x := x + 1
30       /* round 2: both */
31       for each slot y < x not ready to execute:
32         poll $buf_j[y].P[j]$ of other replicas $s_j$
33         on $buf_j[y].P[j]$ is set:
34   *       if $buf_k[y].P[l]$ and compare $buf_j[y] = buf_k[y]$
35             set $buf_k[y].P[j]$
36         on $f+1$ P-flags are set:
37   *       set $buf_k[y].C[k]$
38   *   /* round 3 */
39   *     poll $buf_j[y].C[j]$ of other replicas $s_j$
40   *     on $C[j]$ and $f+1$ P-flags set in $buf_j[y]$:
41   *       set $buf_k[y].C[j]$
42   *     on $f+1$ C-flags are set in $f+1$ replicas
43           $buf_k[y].A[k]$ /*mark $y$ as ready to execute*/
44       /* consensus reached */
45       for each slot y < x:
46         if all slots z < y are executed or skipped
47            and ready to execute($y$)
48           $result$ := execute $buf_k[y].req.m$
49           /* reply to client */
50           $c_i.reply[k].m := result$
51           $c_i.reply[k].seq := buf_k[y].req.seq$
52       /* wrap around */
53       if all slots y < x are executed
54         and x = buffer_length
55         compute checkpoint $C$
56         store $C$ in write-once memory and set $P$ flag
57         if $f+1$ matching checkpoints are written
58           reset flags, buffers
59             and the previous checkpoint; $x := 0$
```

FIGURE 4. Normal Phase, Checkpoint and Buffer Reset

```
60       /* replica $s_k$ */
61       on timeout or error:
62         for each slot y $\leq$ [x]
63           set all E bits for unset agreement bits(*)
64           poll $buf_j[y]$ of other replicas $s_j$
65   *       let $c$ be the number of \emph{wo} memories
66   *          that have crashed
67           wait until either $f+1-c$ replicas have pre-
68              pared the request or $f+1-c$ have reached
69              an error state with $\geq f+1$ E-flags set
70           in the former case
71   *         identify request $m$ such that $m$ matches
72   *            the request in the buffers of $\geq f+1-c$
73   *            replicas that have prepared this request
74             execute request // (ln. 46-51)
75           otherwise skip the slot by setting $buf_j[y].AE$
```

FIGURE 5. Error handling

VII. ERROR HANDLING

Once healthy replicas time out they no longer modify their acceptance flags. Instead, they set the error flags corresponding to all acceptance flags (*AE* flags) not yet set in all slots y that have been proposed, but not yet completed, including in all slots for which the current leader is responsible. We denote the latter by $[x]$. The *wo* memory detects if the *A*-flag or its corresponding error flag *AE* is set in $f + 1$ replicas and will trigger the equivalent of the operation from Line 63 in all *wo*-memories to ensure replicas can no longer change flags after the majority timed out. Replicas will not engage into actually processing this timeout before either $f + 1 - c$ replicas have prepared the request or $f + 1 - c$ reached an error state where the tri-state nature of flags prevent them from preparing it later. Here, c is the number of *wo* memories that have crashed.

Similar to MinBFT, we define as necessary condition for a replicas to have prepared a request that it has set $f+1$ of its P-flags, which resembles *iBFT*'s notion of having received $f+1$ prepare messages. However, we consider a replica as prepared only if it either completed executing the request (i.e., if it has $f + 1$ $C-flags$ and the A-flag set as well, respectively only the A-flag for the no-crash case), or if it has timed out and set all error flags for the agreement flags that remained unset and if in this state it has set at least $f + 1$ P-flags. If replicas set a P-flag, the trusted *wo* memory implementation prevents them to also set the E-flag. It is important to require replicas to have timed out before considering them to be prepared in a state less advanced than all flags set that are required for execution since replicas need to independently reach the same conclusion whether or not a request should be processed.

Replicas execute those requests for which they find that $f + 1 - c$ replicas having prepared this request (Lines 70–74). They skip executing this slot if $f + 1 - c$ replicas have reached an error state from which they cannot later prepare it (Line 75). Since the leader's *wo* memory might have crashed, this request may reside as a copy in another replica's buffer. Lines 71–72 identify this request.

VIII. CHECKPOINTS AND RESET

Once all slots are used up, replicas have to reset the buffer before they can proceed. Without such a reset, slots, which now have flags set, would not be writable due to *wo* memory preventing overwrites. However, there are three inherent race conditions when resetting buffers:

1) A faulty replica may prematurely agree to reset the *wo* memories before the checkpoint is stable;
2) A replica may vote to reset a buffer that has just been reset; and
3) A lagging, but otherwise healthy, replica may resume in a slot after the other replicas have reset all *wo* memo-

ries. In this case a faulty replica may exploit the lagging replica to replay an old request that the lagging replica was about to handle.

We avoid the first by requiring healthy replicas to first agree on a checkpoint and wait for this checkpoint to stabilize before agreeing to reset the *wo* memories. Checkpoints are written to write-once memory as well, using double buffering to always have a valid checkpoint in place. Checkpoints include a version number to denote which of the buffers holds the most recent checkpoint. Like for requests, *wo* memory prevents modification of completed checkpoints by setting a corresponding P flag (Line 57). Once a healthy replica detects $f + 1$ matching checkpoints, it agrees to reset the buffers in all replicas, including the now old checkpoint.

The second race is in fact an instance of the first since, without further precautions, agreeing to reset after the reset already happened translates into prematurely agreeing to the reset in the next round. We shall use the same mechanism to prevent the second and third race condition: We use one additional flag RF in the bitfields to denote that a reset has just happened. RF is checked when writing *wo* memory or when setting flags to prevent any modification of the tag-based *wo* memory device due to ongoing operations. Instead, these operations will fail, leaving the device in the state after reset, which allows the replica to recover from this situation. Moreover, RF is checked when agreeing to reset *wo* memory. The agreement is ignored when RF is set.

In consequence of the above, after each *wo* memory write or set flag operation and after reset in Line 57, the replica checks whether the device has just undergone reset and reacts to this by clearing all RF flags, loading the most recent checkpoints and resuming from this checkpoint and an empty buffer. We have omitted these checks from the pseudo code for better readability. RF flags are the only flags that can be reset by the writing replica, but only by this one. As indicated above, the most recent checkpoint is the one that received $f+1$ agreement and that has the higher version number of the two checkpoint slots.

IX. RESET

Obviously, replicas consume *wo* memory space over time as they use it to handle requests. Therefore, once the available buffer space is used up, replicas have to reset *wo* memory to clear all tags before they can resume processing requests. We shall align this reset with the writing of a checkpoint and store the latter as well in *wo* memory. Double buffering alternates between checkpoint buffers and ensures that the latest checkpoint always remains intact.

Single handed or premature reset would allow replicas to equivocate, by resetting and overwriting a field after another replica has introspected it. We therefore make reset a consensus operation and require $f + 1$ replicas to agree before tags are cleared. The fact that a reset has just happened is recorded by setting reset flags RF, which are checked together with the remaining bits of the bitfield, but which can be cleared by the replica to continue writing to the device. We shall return to the necessity to synchronize checkpoints and resets in Section VIII.

Several implementations of the above reset functionality are conceivable. For example, replicas could enter a trusted execution environment (TEE), e.g., enclaves, and implement reset by waiting for $f + 1$ replicas to enter their TEE before clearing *wo* bitfields and strings through normal writes or through a dedicated interface. Obviously, the permission to perform these operations must be restricted to the TEE.

Alternatively, reset could be implemented as a second device, similarly to write-once memories, collecting the intention to reset in a bitfield with one bit per replica. The device resets all *wo* memories (clearing bits and making strings writable again), as described above, after $f + 1$ replicas agree by setting their reset bit. Naturally, reset must as well be part of the RCB. We have implemented this option for our evaluation, due to the high costs of entering and leaving TEEs.

It is of course possible to implement a reset device per core, capable of only resetting this core's write-once memory instead of a general one, to avoid a single point of failure.

X. TRUSTED COPY

iBFT allows reaching consensus on an operation, but does not perform this operation by itself. However, to act in a consensual manner, state must be updated, including configurations and privileges, as described in Gouveia et al. [28]. In the following, we introduce a mechanism, which complements *iBFT* to safely reconfigure privileges and update critical data through a trusted copy operation.

iBFT reaches consensus out of place, that is in the write-once memories and not in the place where the platform expects the data (e.g., in the processor's page tables or page-table base register). To update these locations, we introduce trusted copy as an operation to transfer data from write-once memories to such a location, but only if (1) all replicas have agreed on the operation, (2) if all previous operations are applied or have been skipped since agreement was not reached for them, and (3) only once (that is, once the data is copied, no further copy operations are allowed for the slot containing this data until *wo* memory is reset).

To perform the copy, we now interpret the message m in a slot slightly differently and introduce an additional tag to mark if data was already copied. Unlike the previous tags, this tag is only writable by the copy operation and remains set until reset. For trusted copy, we divide m into a destination address, a size field and a data field with the semantics that data of the mentioned size should be copied to the mentioned destination. Replicas agree on this triplet and then leave it to any replica to perform the copy, which will succeed under the above mentioned conditions.

Figure 6 lists the simple pseudo code for this procedure. After checking whether agreement has been reached (i.e., the ready-to-execute flag is set by $f + 1$ replicas for a given slot l) or whether the previous slot is ultimately skipped, since no agreement is reached, the copy operation validates that the previous slot is marked as executed. It then marks

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

```
TC $tc_i$, l:
if $l$ is marked not executed
 and $l-1$ is marked executed
  mark $l$ as executed
  if $l$ marked ready to execute on $f+1$
    $dest.[req.m.addr] := buf_l[x].req.m.data$
```

FIGURE 6. **Trusted Copy Operation.**

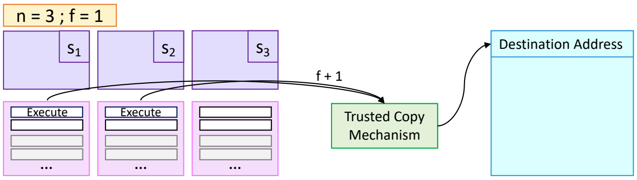


FIGURE 7. **Representation of the trusted copy mechanism, copying an agreed-upon request to the designated destination address.**

l as executed and if agreement has been reached copies the data to the mentioned address. Note that this entire operation must be executed atomically with regard to other trusted copy operations, since otherwise faulty replicas could force out-of-order updates. Hardware implementations can achieve that by performing only one trusted copy operation at a time. Figure 7 illustrates the trusted copy mechanism, copying an agreed-upon request to the designated destination address.

As with *wo* memory, the implementation of the trusted copy is simple enough and can be trusted not to fail (supported for example through formal verification). It can further be provided in a redundant manner to guarantee continued operation in the case of a crash and the memory itself (plus the memory controller) may as well have some form of redundancy depending on the desired fault model. Recall that the failure of a *wo* memory simply means the associated replica is now considered faulty. The trusted copy is not implemented for each replica, but instead an instance that collects results. Therefore, its level of redundancy is not dependent on the value of n.

XI. IMPLEMENTATION DETAILS

To further clarify *iBFT* and *wo* tagged memory, we shall describe in this Section the implementation details of the latter in 1) the ZC702 FPGA board and 2) emulation version, for our proof-of-concept.

We turn blocks of memory into *wo* tagged memory by using sticky tags implemented as a hardware accelerator slave device that `AND`s the write enable signal of a memory controller, with the devices tag verification, allowing writing tags only if they are clear and allowing writes to memory only if tags indicate to the *iBFT* protocol that agreement has been reached.

We consider and evaluate two implementations of write-once memory:

- 1. Using tagged memory hardware devices (implemented on an FPGA for proof-of-concept);
- 2. Using an emulation of microcode-based atomic operations to conditionally set bits in bitfields or write parts of the string, provided the string is not marked read.

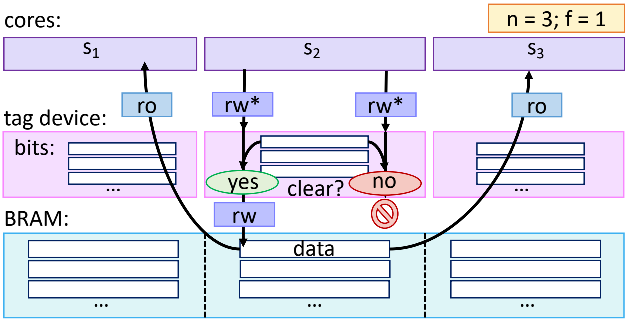


FIGURE 8. **Implementation of *wo* memory as a combination of an AXI slave tag-mem device and a standard BRAM block.**

A. HARDWARE-BASED

To evaluate the first variant, we have implemented *wo* memory as a combination of a standard per-replica block RAM (BRAM) area to hold *wo* strings and an AXI Lite[11] slave device for implementing *wo* bitfields (one per buffer), as shown in Fig. 8. The slave device interposes writes and prevents overwriting strings that are marked as ready by setting any one of the bits in the corresponding bitfield. Moreover, it prevents the replica from clearing bits by `AND`-ing updates to the inverse of the bits that are already set (both error and agreement bits), prior to `OR`-ing them to the stored value. We denote this in the figures as restricted read/write permissions ($rw*$). Peer replicas obtain direct read-only access to the bitfields and string buffers.

Write enabling writes in the memories is still done by the regular memory controller, however, the enable signal is `and`-ed (logic-wise, with no code) with an enable signal produced by the tagged-memory accelerator upon receiving (from the memory controller) the address that is to be written. This hardware logic will evaluate the bitfields set for that address and determine whether the write is allowed. If it is not, it will output a write enable signal of 0, which `and`-ed with the 1 from the memory controller will still prevent the write. The write-once memory concept is not merely a block of memory (e.g., BRAM), but a simple hardware abstraction that contains memory space (e.g., BRAM) and an associated logic for checking flags (tags) and storing them (`t-mem` in Fig. 1) for an incoming address that will determine the write enable output for writing on the memory block. Alternatively, the tags could be implemented in a custom memory controller itself.

---

[11] The Advanced eXtensible Interface (AXI), part of the ARM Advanced Microcontroller Bus Architecture 3 (AXI3) and 4 (AXI4) specifications, is a communication interface for on-chip communication. AXI interface IP blocks are common in block designs for Xilinx FPGAs, such as the one we use in our implementation (Zynq ZC702).

B. SOFTWARE-BASED

This variant is a trivial microcode exercise by constraining the operations that can write the otherwise read-only memory pages used for *wo*-memory. In fact, aside from this enforcement, contemporary architectures, such a Intel x86, can already emulate *wo* memory in a performance-preserving manner. Write-once bitfields are written exclusively by bit set operations (e.g., a generalized `lock; bts` as in x86, i.e., atomic bit test and set, but for tri-state flags). Write-once character strings are written by atomic compare and swap, where compare checks for a specific value reserved to denote an empty buffer. Of course, full microcode access would also allow for cache-lock protected multi-address conditional writes, checking the bitfield and writing conditionally to the bits being clear.

By restricting which operations can be executed on write-once memory blocks (e.g., through a memory type or page permission flag), it is possible to utilize standard memory subsystems for implementing *wo* memory. The *wo* bitfields can be constrained to only allow atomic bit-set operations (again, a generalized `bit_test_and_set`), checking both error and agreement bits, and the wo strings can be realized by reserving one value (e.g., `exp = ~ 0UL`) to denote writable words, and by writing with atomic `compare_exchange(dest, exp, value)`. Cache locks are one common way to implement atomic read-modify-write instructions in modern processor architectures. More fine-grain control over these locks opens further, more direct ways of implementing write-once semantics. For example, one could make writes to strings conditional to tags in the bitfield being clear. The above mechanisms do not prevent caching write-once memory locations. Aside from requiring atomic operations to write these locations, microcode-based implementations therefore incur no extra overhead. However, the RCB of this variant necessarily includes all hardware components that are required to execute instructions atomically (i.e., all processors, caches and the used fragment of the memory subsystem). Our hardware-based variant further reduces the RCB.

XII. EVALUATION

iBFT's goal is efficient FIT for on-chip systems. As briefly discussed in earlier sections, we envision MPSoCs, chiplets, or a combination of both, enhanced with FPGA fabric for custom, simple and easily-verifiable accelerators such as, in this case, tagged memory.

As proof-of-concept of our solution we have implemented the architecture depicted in Fig. 8 on a Xilinx Zync ZC702 FPGA configured with 3 MicroBlaze cores (running at 100MHz) and AXI busses to connect to the memory controller and our tagged memory and reset devices. We run *iBFT* on each of the MicroBlaze cores (in a final solution, it could instead run on hard cores, as part of the MPSoC) and measured its performance in an $f = 1$ setting. For this variant (FPGA), we evaluated exclusively the setting $f = 1$ due to FPGA resource constrains[12]. We have used an AXI Timer and Interrupt Controller for time measurement and utilize the Xilinx Vivado post-implementation reports for area usage and power analysis.

Our measurements focus on two scenarios: i) agreement with all replicas participating and ii) catch-up with one replica remaining unresponsive while the remaining replicas reach agreement to then catch up with the progress they made. Replicas do not write checkpoints or wrap around buffers in this scenario. We evaluate both *wo* failure by crashing and the case where no further guarantees are provided in case *wo*-memory fails.

For the second variant, the emulation of microcode-based *wo* memory modifies *wo* bitfields with atomic `OR` instructions (`lock; orq`) and *wo* strings with atomic compare exchange instructions (`lock; cmpxchgq`), which check for $\sim$ `0UL`. The implementation always writes the complete string buffer for a single slot to prevent faulty replicas from appending to shorter prefixes. Reads are through arbitrary instructions. The emulation described above exhibits correct performance characteristics, but does not prevent writes through other instructions or unaligned writes with the above instructions. This behaviour can be easily retrofitted through microcode instructions. We evaluated performance on the cache-based x86 emulation, with up to $n = 2f + 1 = 13$, to tolerate up to $f = 6$ faults.

A. PERFORMANCE

1) Performance of Microcode Emulation

All figures plot the mean latency of request handling in cycles as experienced by clients (i.e., the time between issuing a request and receiving $f + 1$ matching responses) (y-axis), for an increasing number of tolerated faults (x-axis). Cycles can be converted into microseconds by dividing cycles by the used frequency (100MHz). For instance, 1000000 cycles corresponds to 10000 microseconds or 0.01 seconds.

Fig. 9 shows the time to agreement (Scenario 1), i.e., normal case execution, whereas Fi. 10 shows the two cases of Scenario 2, that is, normal-case operation of $n-1$ replicas and time for the late replica to catch up. The figures identify the graph bars corresponding to the *iBFT* cache-based version of *wo* memory on x86 in the situation where *wo* memories can crash, side-by-side with the corresponding FPGA hardware implementation (which we shall discuss next). These results are as well compared with a shared memory-based implementation of MinBFT.

Catch up in MinBFT is implemented as the lagging replica receiving the messages sent by the other replicas (by reading their message buffers) and processing the request as usual.

As can be seen, *iBFT* is roughly 10 times faster than MinBFT when reaching agreement (16, respectively if *wo*

---

[12] Note that we refer specifically to Zynq ZC702 resource constraints, where we could only instantiate up to 4 MicroBlaze cores plus the corresponding tagged memory devices, block memories and AXI interfaces, bringing the maximum possible f to 1 ($n = 3$). Other, modern FPGA boards will allow for more replicas to coexist.

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

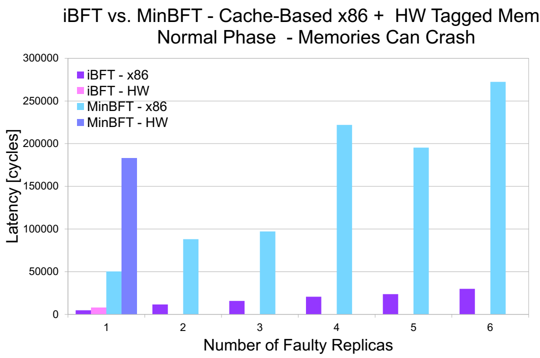


FIGURE 9. **Latency of normal-case operation (in cycles), comparing cache-based and the tag-mem variant of *iBFT* against MinBFT on the same platform. *wo* memories can crash.**

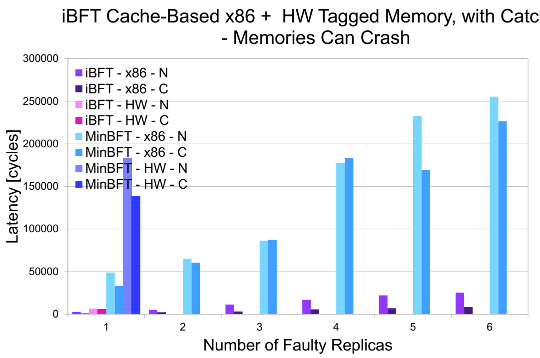


FIGURE 10. **Latency of normal case operation (N) with one late replica and catch up (C) of this replica. *wo* memories can crash.**

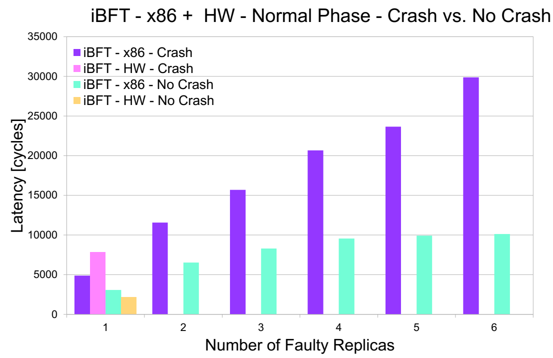


FIGURE 11. **Comparing normal case *iBFT* when *wo* memories can crash vs. when they are assumed not to not crash.**

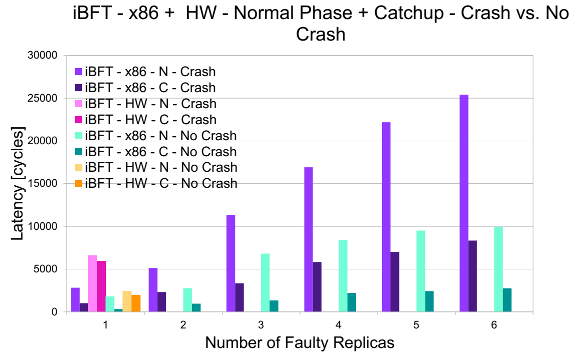


FIGURE 12. **Comparing normal case *iBFT* plus catch up when *wo* memories can crash vs. when they are assumed not to not crash.**

memories do not crash), which we attribute mostly to the costs of HMAC computation and validation, but in a significantly smaller part also to the larger message sizes that origin from having to transmit up to two HMACS (for commit). The optimization of *iBFT*, which allows lagging replicas to catch up to the progress of the leading ones, proved effective, by requiring only 1019 cycles on average for $f = 1$ (324 respectively for the no-crash version), with a linear increase for higher f.

We consider also a model where *wo* memories do not crash. Figs. 11 and 12 represent a comparison of both environments: where memories can crash and where memories do not crash.

In an environment with no *wo* crashes, the reader may notice a stabilization of the latency with the increasing number of replicas participating. The ratio of reads (to introspect peers) versus writes (to update replicas' own state) increases. The cache coherence protocol executes these reads in parallel, which leads to the smoother slope in the graph. Cross hyperthread[13] pre-fetching further improves performance.

It is also relevant to mention latency numbers can slightly vary depending on which replicas are late. Since replicas can proceed once they find $f + 1$ occurrences of the information sought after, and since introspected replicas start sequentially from the replica with the lowest ID to the one with the highest, if there is no late replica in the first $f + 1$, latency will not be affected by non-consecutive reads of late replicas' state. For the shown evaluation we let the late replica always be the one with highest ID, meaning it does not interfere with normal-case operation. Giving late replicas low IDs would slightly increase latency by a few cycles corresponding to introspecting the late replica.

2) Performance of FPGA Implementation

For the following discussion, let us notice that writing a tag-mem device register (i.e., a word of 32 bits) in the shared BRAM block requires 65 cycles. This corresponds roughly to the time required to reach the shared cache (L3) on x86. This translates into 99 cycles for setting flags and 106 cycles for reading.

Like above, Figs. 9 and 10 show the performance for the two scenarios (normal-case only and normal-case plus catch up) for *iBFT*, and Figs. 11 and 12 represent the no-crash case.

As can be seen, the previous results from the emulation-based *wo* memory implementation are confirmed. With a factor of 23.24 (83.09 respectively if *wo* memories do not crash), *iBFT* is, on average, almost one order of magnitude

---

[13] Cores can support hyper-threading, the implementation of which would be assumed part of the RCB in our cache-based variant. In our second variant, the core as a whole is considered the fault containment domain. That is, even though the core may have multiple hardware threats, there can only be one replica on this core.

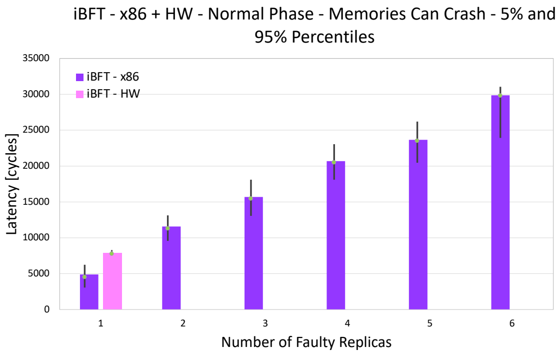


FIGURE 13. **Mean values (bars) together with the 5% and 95% percentiles for both versions of the *wo* memory in *iBFT*.**

	Tagged Memory		Reset Device
	wo/ AXI	w/ AXI	w / AXI
Slice LUTs	1970	2034	58
Slice Registers	2048	2204	145
F7 Mutexes	392	392	0
F8 Mutexes	94	94	0

TABLE 1. **FPGA resources required by the tagged memory device (without / with AXI interface) and the reset device.**

faster than shared-memory MinBFT and almost two orders of magnitude faster in the no-crash version. However, the percentiles are much closer to the average times (see Figure 13), which on one side is due to the higher determinism of tightly-coupled memory accesses over coherent caches, with frequent bounces when polling shared data for state changes. However, it also indicates higher best case costs due to the inability to cache state. Catch up remains relatively fast with a 5927 cycle latency on average (1972 respectively for the no-crash version).

iBFT's latency clearly demonstrates the benefit of constructing hybrid BFT-SMR protocols specifically for tightly coupled systems. The value of introspection is confirmed, in particular when replicas have to catch up to the progress of their peers. Of course, we are naturally introducing overhead in comparison to non-replicated operation, but with added fault tolerance and resilience. Considering the costs of fully replicating the whole system (e.g., ECU), *iBFT* offers a safety advantage without greatly increasing replication costs.

B. COMPARISON WITH MINBFT

To further exemplify why simply implementing a SotA BFT protocol on-chip can be too costly, we have created a shared memory implementation of the seminal MinBFT [47] and evaluated its performance in the same design setting as *iBFT*.

iBFT is, on average, almost one order of magnitude faster than shared-memory MinBFT and almost two orders of magnitude faster in the no-crash version.

	Power (W)
1 Core design	1.602 W
3 Core design w/o tag and reset devices	1.748 W
3 Core design w/ tag and reset devices	1.760 W

TABLE 2. **Total on-chip power in watts (W) of a baseline design with 1 core, *iBFT* architecture design with 3 cores but without the tag and reset devices, and *iBFT* with the tag and reset devices.**

C. AREA USAGE

Tab. 1 shows the FPGA resources of the (post-synthesis) implementation of the tagged memory and reset devices. For the tagged memory device we present two values, the resources for tagged memory logic (i.e., the logic implementing the flags and their write-once property) alone and for tagged memory logic with AXI interface included. Since tagged memory is implemented in our proof-of-concept as an AXI bus slave, it requires logic to interface with this bus, thus consuming further resources.

As expected, the hardware overhead, in terms of resources, of tag-based *wo* memory is dominated by the resources required for the BRAM block itself. Costs for the tag-mem IP are negligible.

D. POWER ANALYSIS

Tab. 2 shows the total on-chip power required by an implementation with and one without the tag and reset IPs. This is the power analysis reported by Xilinx Vivado, based on simulation and constraint files, with default settings, upon implementation of the design.

The additional power consumption added by tagged memory is negligible, representing only a 0.2 W increase in relation to the whole FPGA design with the 3 MicroBlaze cores. We compare also with a baseline design of only 1 core, to show the increase in power consumption by using replication is only of 0.146 W.

All designs include the core plus AXI interconnect, BRAM controller, BRAM and PS Core. The PS core is not needed for the protocol and is just used in the proof-of-concept for interacting with the programmable logic (FPGA) and initializing it. The core is, however, responsible for 96% of the total predicted power consumption.

XIII. BACKGROUND

A. FAULT-TOLERANCE

Fault tolerance (i.e., constructing a system in such a way that it retains the ability to sustain correct operation despite the presence of faults) has been used for years, commonly in the form of dual (DMR) or triple modular redundancy (TMR) to replicate, e.g., critical control tasks. Modular redundancy refers to the multiplication of system components, providing redundancy should one fail. These 'cloned' components usually work in parallel (often in lockstep) with the same state so as to make sure at least one keeps operating and achieves the correct result. However, the lack of flexibility limits the extension to general systems, namely as it usually translates to replicating whole subsystems. Additionally, component isolation and diversity is not always taken into consideration, leading to fault propagation and common mode failures.

More complex solutions target low-level management software, such as operating systems or hypervisors with the intent of improving their fault resilience. However, typically they protect applications [2]–[4] or specific OS subsystems [5]–[8] and/or only from accidental faults [40], not malicious attacks or arbitrary behaviour. Efforts for providing whole-OS fault tolerance include [16]–[22]. Nevertheless, the complexity of these recovery kernels is comparable to that of a small hypervisor. This complexity makes the likelihood of residual faults or vulnerabilities non-negligible and turns this software layer into a single point of failure.

Byzantine fault-tolerant state machine replication (BFT-SMR) [46], [47], [49], [50], paired with rejuvenation [51], [52] and diversification [53]–[58], although traditionally applied in client-server setups, is one combination of techniques that bears the promise of automatic and unattended resilience against both faults and intrusions also in on-chip scenarios. However, while extensive FIT work has been done in the traditional distributed systems realm, little effort has been put into the emerging (multiprocessor) systems-on-chip (MPSoCs) or chiplet architectures that, having their own on-chip network, are starting to resemble tightly-coupled, single-die distributed systems, as pointed out in [28], [59] and [60]. Moreover, the costs of BFT-SMR solutions are prohibitive on the latter due to the use of costly, yet necessary, cryptographic operations for transferable authentication of a replica's messages, and for the amount of messages exchanged. Both hinder achieving high performance and increase power consumption.

B. ARCHITECTURAL HYBRIDIZATION

Architectural hybridization [47], [49], [50], [61], [62] utilizes the inclusion of trusted-trustworthy components, which fall under a distinct fault model from the rest of the system and which are considered more resilient. Examples include MinBFT's USIG [47] and CheapBFT's CASH [62], both implementing trusted counters, and A2M [63] and TrInc [64], which provided a trusted message log or its hash. In the realm of BFT, this allows reducing the number of required replicas to safely reach agreement from $n = 3f + 1$ to $n = 2f + 1$, where f is the fault threshold.

C. TIGHTLY-COUPLED SYSTEMS

FIT solutions have been designed for tightly-coupled systems before. For example, replica coordination support was first incorporated into a hypervisor in [65] and the crash-fault tolerant protocol Paxos [66] was implemented as a Linux kernel module in [67]. Support for replication in microkernel-based systems was achieved in [68] and [69] uses non-blocking consensus to tolerate up to one crash fault. More recently, [40] explored tightly-coupled redundant execution on replicated hardware in the context of accidental faults.

In a different approach, [70] leverages RDMA in the crash fault-tolerant system Mu to bring SMR performance down to microsecond scale, and also for BFT [71]. Nevertheless, Mu relies on changing RDMA write permissions to allow the leader to directly write into follower logs, which involve the OS (e.g., manipulating page tables) and could induce significant costs (e.g., through TLB flushes).

D. FPGA SECURITY

Although our work targets systems-on-chip in general, whether in the form of ASICs, chiplets, FPGAs or a heterogeneous combination them, it is relevant to discuss FPGA security measures, namely given our proof of concept is implemented in one. Multiple strategies have been proposed and used regarding FPGA security which, in turn, have an effect on the system's safety as well as resilience. For instance, encryption and authentication [72]–[74] have been adopted to protect bitstreams against intellectual property (IP) piracy [75], trojan insertion, data leaks, etc. Different solutions for key storage and protection have also been proposed, such as the use of physically unclonable functions (PUFs) [76]. However, (i) these techniques mostly target the protection of hardware, i.e., bitstreams and the keys used to decrypt them, not the software running on cores; and (ii) even such mechanism have been the target of attacks [77]–[80].

E. EQUIVOCATION

In classical distributed consensus protocols, replicas send messages to each other through an Ethernet connection. Taking the example of PBFT [46], once a replica receives a message from another with a certain sequence number, it will ignore further messages from the same replica with the same sequence number, forbidding the sending replica from "changing its mind" about the request being voted upon. Inside an MPSoC, however, and if data is written in memory, replicas can simply read the votes of others. As such, one must be cautious about an important detail: the time at which a replica reads the memory where the proposals are stored.

Sadly, a key factor of FIT solutions — namely, authentication — when Byzantine behaviour is expected, becomes problematic in the context of MPSoCs. All practical BFT protocols rely on the presence of authentication and, thus, cryptographic operations in order to ensure replicas do not impersonate others or lie about their votes. PBFT, for instance, relies on digital signatures, requiring that requests and every message passed among replicas are authenticated with the utilization of message authentication codes (MAC), the keys of which are changed during recovery to avoid impersonation if an attacker learns the MAC keys. In MinBFT [47], for instance, the trusted-trustworthy device USIG is in charge of signatures and provides two simple operations `create UI` and `verify UI`. Every message generated by a USIG is tagged with a certificate called UI (unique identifier), containing an ID (the replica's unique identifier), a monotonically increasing counter value and a signed hash of the message; and serves the purpose of uniquely identifying messages.

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3484013

These generated signatures are then verified in other replicas' USIGs.

Cryptography costs, although perfectly acceptable in the context of distributed systems and their BFT implementations, given their fair performance ratio considering Ethernet message passing costs, would not be suitable in on-chip scenarios, since local transfer operations and cross-tile NoC bus costs are in the microsecond to nanosecond domains.

Close to native communication latency therefore requires abandoning cryptography and, with this, transferable authentication [81]. It is, therefore, impossible to distinguish a scenario where the sender of a message falsely sends (i.e., writes) some information from one where the receiver (i.e., reader) modifies it. Consensus without transferable authentication was first investigated by Lamport in the oral messages (OM) protocol [82], where an impossibility to diagnose errors, and hence recover from situations where replicas could lie inconsistently to others (i.e., equivocate), was identified. In other words, replicas lose the ability to prove the origin of messages once this message leaves the originator's state.

To circumvent this impossibility, we rely on architectural hybridization [47], [49], [50], [61], [62], i.e., the introduction of a trusted-trustworthy component, and present our tightly-coupled BFT-SMR protocol — *iBFT*, with the aid of a write-once tag trusted-trustworthy component that serves as the means to avoid equivocation (see Section XIII-E).

XIV. CONCLUSIONS AND FUTURE WORK

This paper tackled the predominant issues in FIT for on-chip systems, attempting to devise a solution that relies on no single-point-of-failure software layer and no synchrony reliant protocol, tolerates arbitrary faults and has acceptable performance for tightly-coupled environments. We introduced the FIT protocol *iBFT*, a BFT-SMR protocol design with such a goal in mind. We showed how *iBFT* circumvents a well known impossibility identified by Pearson et al. for BFT-SMR protocols that cannot rely on transferable authentication, which is the case for tightly coupled BFT-SMR protocols if they want to remain close to the performance of the replica connecting communication medium: the on-chip networks of multi- and manycore systems and the shared memories they connect. We introduced trusted-trustworthy hardware-based components to establish the notion of write-once memory and have shown that, with these components, Directions for future work include applying dynamic reconfiguration to the tag trusted devices and potentially the cores as well for systems deployed entirely on Programmable Logic fabric.

REFERENCES

[1] Tyson T Brooks, Carlos Caicedo, and Joon S Park. Security vulnerability analysis in virtualized computing environments. *International Journal of Intelligent Computing Research*, 3(1/2):277–291, 2012.

[2] Alex Depoutovitch and Michael Stumm. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 181–194. ACM, 2010.

[3] Cristiana Bolchini, Matteo Carminati, and Antonio Miele. Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems. *J. Electron. Test.*, 29(2):159–175, 2013.

[4] Dmitrii Kuvaiskii, Rasha Faqueh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted Fault Tolerance. In *11th European Conference on Computer Systems (EuroSys)*, pages 1–17, 2016.

[5] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. *Trans. Storage*, 6(3):11:1–11:30, 2010.

[6] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, 2006.

[7] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 4–4. USENIX Association, 2006.

[8] Kevin Elphinstone and Yanyan Shen. Increasing the trustworthiness of commodity hardware through software. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[9] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *37th International Symposium on Computer Architecture (ISCA'10)*, 2010.

[10] Roger M. Needham and Maurice V. Wilkes. Domains of Protection and the Management of Processes. *The Computer Journal*, 17(2), 1974.

[11] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Architectural Support for Programming Languages and Operating Systems*. ACM, 2016.

[12] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25. ACM, 1995.

[13] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI*, volume 8, pages 225–240, 2008.

[14] Neeraj Gandhi, Edo Roth, Brian Sandler, Andreas Haeberlen, and Linh Thi Xuan Phan. REBOUND: Defending Distributed Systems Against Attacks with Bounded-Time Recovery. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 523–539, 2021.

[15] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 457–468. IEEE Press, 2014.

[16] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proceedings of the Sixth European Dependable Computing Conference*, EDCC '06, pages 3–12. IEEE Computer Society, 2006.

[17] Ruslan Nikolaev and Godmar Back. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132. ACM, 2013.

[18] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability Through Operating System Structure. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 59–72. USENIX Association, 2008.

[19] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. 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, pages 49–60. ACM, 2009.

[20] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36, 2016.

[21] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth ACM Sympo-*

sium on Operating Systems Principles, SOSP '99, pages 154–169. ACM, 1999.

[22] David Gens. OS-Level Attacks and Defenses: From Software to Hardware-Based Exploits, 2018.

[23] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience. In German Society of Informatics, editor, *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, 2013.

[24] William J Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th annual design automation conference*, pages 684–689, 2001.

[25] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[26] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[27] Michael Engel and Björn Döbel. The reliable computing base: A paradigm for software-based reliability. In *Workshop on SOBRES*, 2012.

[28] Inês Pinto Gouveia, Marcus Völp, and Paulo Esteves-Verissimo. Behind the Last Line of Defense: Surviving SoC Faults and Intrusions. *Computers & Security*, 123:102920, 2022.

[29] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices. *Sensors*, 21(7):2329, 2021.

[30] BP Prabahar and BE Edwin. Survey on virtual machine security. *International Journal of Advanced Research in Computer Engineering Technology (IJARCET)*, 1(8):115–121, 2012.

[31] Louis Turnbull and Jordan Shropshire. Breakpoints: An analysis of potential hypervisor attack vectors. In *2013 Proceedings of IEEE Southeastcon*, pages 1–6. IEEE, 2013.

[32] Ammarit Thongthua and Sudsanguan Ngamsuriyaroj. Assessment of hypervisor vulnerabilities. In *2016 International conference on cloud computing research and innovations (ICCCRI)*, pages 71–77. IEEE, 2016.

[33] Dina Zoughbi and Nitul Dutta. Hypervisor Vulnerabilities and Some Defense Mechanisms, in Cloud Computing Environment. *International Journal of Innovative Technology and Exploring Engineering*, 10:42–48, 2020.

[34] KA Scarfone. *Guide to security for full virtualization technologies*, volume 800. DIANE Publishing, 2011.

[35] Salman Iqbal, Miss Laiha Mat Kiah, Babak Dhaghighi, Muzammil Hussain, Suleman Khan, Muhammad Khurram Khan, and Kim-Kwang Raymond Choo. On cloud security attacks: A taxonomy and intrusion detection and prevention as a service. *Journal of Network and Computer Applications*, 74:98–120, 2016.

[36] Ying C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307. IEEE, 1998.

[37] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. Airbus fly-by-wire: A total approach to dependability. In *Building the Information Society*, pages 191–212. Springer, 2004.

[38] Luigi Mancini. Modular redundancy in a message passing system. *IEEE transactions on software engineering*, 1:79–86, 1986.

[39] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[40] Yanyan Shen, Gernot Heiser, and Kevin Elphinstone. Fault Tolerance Through Redundant Execution on COTS Multicores: Exploring Trade-Offs. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 188–200. IEEE, 2019.

[41] Paulo Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing: Research and Position Papers*, pages 108–113. Springer, 2003.

[42] Subodha Charles and Prabhat Mishra. A Survey of Network-on-Chip Security Attacks and Countermeasures. *ACM Computing Surveys (CSUR)*, 54(5):1–36, 2021.

[43] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[44] Ulrich Schmid and Andreas Steininger. *Decentralised fault-tolerant clock pulse generation in VLSI chips*. TU Wien, 2010. Patent: US7791394B2.

[45] Edward A Feustel. The Rice research computer: a tagged architecture. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 369–377, 1971.

[46] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *Operating systems design and implementation (OsDI)*, volume 99, pages 173–186, 1999.

[47] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine Fault-Tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[48] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 135–144. IEEE Computer Society, 2009.

[49] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 174–183. IEEE, 2004.

[50] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. BFT-TO: Intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693–715, 2012.

[51] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[52] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[53] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 287–292. USENIX Association, 2008.

[54] Tom Roeder and Fred B Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)*, 28(2):1–54, 2010.

[55] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Os diversity for intrusion tolerance: Myth or reality? In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 383–394. IEEE, 2011.

[56] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291. IEEE Computer Society, 2014.

[57] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014.

[58] Miguel Garcia, Alysson Bessani, and Nuno Neves. Lazarus: Automatic management of diversity in bft systems. In *Proceedings of the 20th International Middleware Conference*, pages 241–254. ACM, 2019.

[59] Inês Pinto Gouveia. Architectural support for hypervisor-level intrusion tolerance in mpsocs. 2022.

[60] Ali Shoker, Paulo Esteves Verissimo, and Marcus Völp. The path to fault-and intrusion-resilient manycore systems on a chip. *arXiv preprint arXiv:2307.01783*, 2023.

[61] Paulo E Verissimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

[62] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308. ACM, 2012.

[63] B. Chun, P. Maniatis S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[64] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

[65] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–11, 1995.

[66] Leslie Lamport. The part-time parliament. *Transactions on Computer Systems*, 16(2):133–169, 1998.

[67] Emanuele Giuseppe Esposito, Paulo Coelho, and Fernando Pedone. Kernel paxos. In *37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2018.

[68] Björn Döbel. Operating system support for redundant multithreading, 2014.

[69] Tudor David, Rachid Guerraoui, and Maysam Yabandeh. Consensus inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 145–156. ACM, 2014.

[70] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[71] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 409–418. Association for Computing Machinery, 2019.

[72] Stephen M Trimberger. Method and apparatus for protecting proprietary configuration data for programmable logic devices, 2003. US Patent 6,654,889.

[73] Stephen M Trimberger and Jason J Moore. Fpga security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[74] Ed Peterson. Developing tamper-resistant designs with ultrascale and ultrascale+ fpgas. *XAPP1098. Xilinx Corporation*, 155:156, 2017.

[75] Adam Duncan, Fahim Rahman, Andrew Lukefahr, Farimah Farahmandi, and Mark Tehranipoor. Fpga bitstream security: a day in the life. In *2019 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2019.

[76] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical unclonable functions and public-key crypto for fpga ip protection. In *2007 International Conference on Field Programmable Logic and Applications*, pages 189–195. IEEE, 2007.

[77] Maik Ender, Amir Moradi, and Christof Paar. The unpatchable silicon: A full break of the bitstream encryption of Xilinx 7-series FPGAs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1803–1819, 2020.

[78] Maik Ender, Gregor Leander, Amir Moradi, and Christof Paar. A cautionary note on protecting Xilinx's UltraScale+ bitstream encryption and authentication engine. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–9. IEEE, 2022.

[79] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249, 2010.

[80] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 44–68, 2018.

[81] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 301–308, 2012.

[82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

INÊS PINTO GOUVEIA Dr. Inês Gouveia is a Postdoctoral Fellow at the King Abdullah University of Science and Technology (KAUST) working on the application of dynamic hardware reconfiguration to resilient systems. She obtained her PhD in 2022 at the University of Luxembourg, where she specialized in hardware support for hypervisor-level fault and intrusion tolerance. After her PhD she worked as a Research Scientist at Intel Labs Germany, looking into the design of a RISC-V safety island. Her Bachelor's and Master's degrees where obtained at the University of Lisbon, with a focus on non-intrusive runtime verification.

RAFAL GRACZYK Rafał Graczyk received BSc (2005) and MSc (2009) in Electronics and Computer Engineering and PhD (2017) in Electronics from Warsaw University of Technology. He's got more than 15 years of experience in dependable systems research and development, focusing on resilience and radiation effects in computer systems. He's been working in various roles in projects at the Space Research Center of the Polish Academy of Sciences, at the Interdisciplinary Center for Security, Reliability and Trust at University of Luxembourg and at companies within the European space sector.

MARCUS VÖLP 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 PhD 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-VERISSIMO 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.