Shield: a Middleware to Tolerate CPU Transient Faults in Multicore Architectures

Mohamed Mohamedin
Virginia Tech
mohamedin@vt.edu

Masoomeh Javidi Kishi
Lehigh University
maj717@lehigh.edu

Roberto Palmieri
Lehigh University
palmieri@lehigh.edu

Abstract—**Multicore architectures are increasingly becoming prone to transient faults. In this paper we present Shield, a middleware to provide transactional applications with resiliency to those faults that can happen anytime during the execution of a processor but do not cause any hardware interruption. Shield is inspired by the state machine replication approach, where computational resources are partitioned, the shared state is fully replicated, and requests are executed by all partitions in the same order. Our results using the Tilera reveal limited overhead with respect to the non-fault-tolerant approaches on most benchmarks, and an average performance gain of 1.54× over traditional byzantine fault tolerance protocols.**

I. Introduction

Data are precious and protecting data integrity is critical to many applications. Multicore architectures form the current technological trend and, in such systems, the problem of tolerating data corruption is complex due to the nature of the underlying hardware [1]. As an example, *soft-errors* [2] belong to the category of hardware-related errors that are difficult to detect or predict. Specifically, they are transient faults that may happen anytime during the application execution. They are caused by physical phenomena [1], e.g., electric noise, which cannot be directly managed by designers. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may behave incorrectly.

In this paper we focus on those soft-errors that occur inside the processor (which we name CPU-TFAULTs). That is because CPU-TFAULTs are random, hard to detect, and can corrupt data – e.g., a CPU-TFAULT can cause a single bit to flip in a CPU register due to the residual charge of a transistor, which inverts its state. Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., an unused register). However, sometimes, the register can contain a memory pointer or a meaningful value. In those cases, the application behavior can be unexpected. An easy solution for recovering from transient faults is a simple application-restart, but for those applications with stringent reliability requirements, which form the focus of this paper, it cannot be acceptable.

We propose *Shield*[1], a software middleware for tolerating CPU-TFAULTs. Shield's basic scheme is inspired by the state machine replication approach (SMR) [4], where computational resources are partitioned, data are replicated across partitions, and application requests are executed on all partitions following the same (previously agreed) order. The goal of Shield is to prevent any data stored in a processor register from being propagated to the system's main memory where the shared state is kept, without being verified as free of corruption. Shield is designed for applications where many threads act on the same shared state and where the data integrity is fundamental (in fact Shield finds its sweet spot on transactional applications). A data corruption can let one thread crash or affect the thread's semantics. Roughly, if an application relies on Shield, a CPU-TFAULT can still let some of its threads crash, but it will not let the shared data be corrupted.

Shield does not protect threads' execution outside transactions, as well as the integrity of the operating system because we consider it as an orthogonal problem, which can be solved employing alternative solutions. As an example, the functionality of the operating system can be protected against CPU-TFAULTs using approaches like [5] or the application's execution can be made reliable by using [6]. However, both these approaches care about avoiding service interruption and none of them address the semantics of the application and its data integrity. Shield guarantees that the shared state is not undesirably corrupted by CPU-TFAULTs.

As in the SMR approach, Shield processes transactions in the same order on all replicated states. This redundant execution isolates any possible propagation of incorrect transition to the memory without being previously certified. The latter operation is performed by a *voter*, a software module that collects the outcome of transactions and delivers the common response (i.e., the majority of replies) to the application. The voting outcome is also used for identifying corrupted computations, and, if so, other correct states are exploited for overwriting the broken parts of the memory. As the other software components of Shield, also the voter is not assumed to be reliable. As we will show later, its design allows it to be resilient to CPU-TFAULTs. In order to guarantee limited overhead with respect to the original execution, which is the performance goal of our proposal, Shield reduces transaction latency by exploiting parallelism and advanced hardware features (e.g., hardware clock and message-passing links).

We implemented Shield in C++. We use in-memory transactional applications for evaluating our proposal. Such applications are good candidates as they are typically deployed on

[1]A short version of this paper has been published in [3].

978-1-5386-1465-5/17/$31.00 ©2017 IEEE

multicore architectures and do not use stable storage to log their actions. Our evaluation focuses on a 36-core Tilera TILE-Gx family board [7], which is a shared memory multicore architecture based on message-passing. Results, using micro-benchmarks and well-known transactional applications such as TPC-C [8] and Vacation from the STAMP suite [9], reveal that Shield ensures fault-tolerance with competitive performance of non-fault-tolerant systems, without paying the cost (both in terms of significant performance degradation and financial cost) of deploying a distributed infrastructure.

Summarizing, the paper makes the following major contributions: *i)* A novel total order protocol that exploits hardware features to reduce the delivery latency; *ii)* A concurrency control algorithm that provides in-order commit with a comparable performance with respect to the out-of-order commit version; *iii)* A comprehensive evaluation study on hardware message-passing-based shared memory architecture.

II. Is Byzantine Fault Tolerance the Solution?

CPU-TFAULTs are transient faults, and that class of faults belongs by itself to the category of *Byzantine Faults* (BF) [10]. A BF is an arbitrary fault that can generate incorrect response or corrupt the system state. BFs include commission and omission faults. Solutions to BF, named also *Byzantine Fault Tolerant* (BFT) protocols (e.g., [11], [12]), are usually designed for minimizing the assumptions on the correctness and trustiness of components composing the execution environment, as well as for being resilient to malicious behaviors. Given that, the answer to the above question is clearly: *yes*, BFT is a solution solving also CPU-TFAULTs but, as also supported by our evaluation study, it is very "pricy".

In fact, deploying a BFT solution would have an impact on the system's performance much higher than what is actually needed for solving the problem of CPU-TFAULT. In addition, BTF solutions often require a physical multi-node distributed system to isolate nodes from each other and therefore avoid the propagation of faults. However, replicating centralized systems for tolerating faults results in significantly degraded performance (e.g., 10–100$\times$). That is primarily due to the costs for remote synchronization and communication that are incurred to ensure node consistency. Also, a distributed architecture comprising of multicore nodes may not often be cost-effective. BFT systems handle malicious client behavior and unreliable networks that can reorder, drop, or corrupt messages. In addition, most BFT solutions require $3f + 1$ nodes to reach agreement and $2f+1$ nodes for the transaction execution in order to tolerate up to f faulty nodes [13]. Shield assumes trusted clients and a reliable network infrastructure, e.g., the bus and most message-passing architectures [7].

Shield is meant to be a software layer that can be plugged into a classical centralized transactional system without deploying a distributed infrastructure or substantially impacting the performance of the original system. We believe that adopting a BFT solution for solving the problem of CPU-TFAULTs would be inaccurate because of the excessive negative performance penalty that the application has to undergo without being actually able to exploit the solution as a whole.

III. System & Fault Model, and Assumptions

We consider a system based on the message-passing abstraction where a set of *nodes*, installed on the same physical hardware, communicate with each other through a reliable FIFO channel (e.g., the bus or the hardware message-passing channel). We consider each node as a group of computing cores on a multicore board. For the sake of clarity while illustrating the proposed algorithms, we assume the presence of a service providing a single monotonically increasing clock. Some message-passing boards (e.g., Tilera [7]) are equipped with such a service through special hardware extensions.

Application's shared state is replicated such that each node accesses its own copy. This way, storing a value from a CPU register to a memory location does not interfere with the work of other nodes, which prevents the propagation of a possible fault to other nodes. To tolerate f CPU-TFAULTs, Shield requires $2f+1$ nodes so that a majority can be formed and the voting procedure can take place.

Shield targets CPU-TFAULTs, which are transient faults that occur inside the processor. Shield does not prevent any value corrupted by a CPU-TFAULT from being stored in memory. Rather, it cares about those values that will actually affect the shared state. As an example, if a node is affected by a CPU-TFAULT, it can still write a corrupted value into its private memory space, but it will never propagate that value to the shared state seen by application threads. Also, as we mentioned above, Shield does not protect threads' execution outside transactions, as well as the integrity of the operating system because we consider it as an orthogonal problem, which can be solved by employing alternative solutions.

Shield works on a single machine, thus it cannot tolerate a crash or a permanent hardware failure of the machine. Asynchronous checkpointing to a stable storage can be used to tolerate such failures. In addition, Shield cannot tolerate deterministic software bugs or incorrect application configurations. A deterministic bug will occur on all nodes and generate the same results. This problem could be solved using other techniques, like diversity (e.g., [12], [14]). However, if the software bug is not deterministic, e.g., concurrency bugs that happen on one node due to the particular schedule of operations, Shield's architecture is still able to prevent the results generated from being propagated to the application.

IV. Shield

A. Overview

At high-level, Shield makes a transactional application resilient to CPU-TFAULTs as follows. It logically partitions computational resources into *nodes*. Each node is composed of a subset of the available cores that co-operate for processing transactions. We adhere to the state machine replication approach [4] where transactions are firstly ordered in a deterministic manner across nodes and then processed by all nodes. This way the shared state is kept consistent at every node without any sharing.

There are several proposals that solve total order in systems prone to faults (e.g., Paxos [15], Mencius [16], 1Paxos [17], Caesar [18], M^2Paxos [19]) but none of them target explicitly transient faults (such as CPU-TFAULTs) and are optimized for deployments where there is an efficient clock-service and communication channels are reliable. In this paper we propose a total order protocol that takes advantage of the centralized deployment and exploits the underlying hardware features to reduce the delivery latency.

At a glance, our ordering protocol involves application threads (because they are physically located together with all other nodes) and it does not rely on a single component to order (e.g., the sequencer). When a transaction is requested to execute by an application thread, it is sent to all nodes and other application threads, together with the current timestamp taken from the clock-service. This timestamp represents a tentative order for executing the transaction (which we name *tn-order*) [20]. To determine its total order, a node must ensure that it receives a request from each application thread with a timestamp that is higher than the one just received. (For this reason we said above that our ordering layer involves application threads.) When a node receives a message from all application threads, it can now safely determine the next transaction to deliver and its total order. As we will show in Section V, our ordering protocol can tolerate CPU-TFAULTs also in the shared clock-service.

An ordering-based concurrency control (ObCC) protocol, running locally at each node, is responsible for processing and committing transactions on the node's private memory. At this stage, the application thread is still not informed about the transaction outcome because an additional *voting* phase (see below) is required. A subset of cores in each node is dedicated for the execution of ObCC. ObCC leverages the tn-order for anticipating the transaction processing. We name this execution as "speculative" because the tn-order is tentative and can be contradicted by the total order. Also, in order to maximize the overlapping of the transaction processing with the establishment of the total order, transactions are processed in parallel (using the tn-order for solving conflicts).

A *voter* is in charge of collecting transaction outcomes from all nodes and returning the majority of them to the application. Even though this approach potentially increases the end-to-end transaction latency because the voter has to wait for a majority of outcomes, nodes are part of the same architecture thus their progress is likely not skewed. Using a lazy concurrency control, where operations are buffered until reaching the commit phase (as in our ObCC), simplifies the comparison procedure of the voter because each transaction keeps track of all its written objects in the *write-set* and all the read objects in its *read-set*. We cannot use hash-based signatures instead of read-set and write-set because hash collisions (when two values map to the same hashed value) can hide faults. The voter is stateless and, as we will show later (Section VII), it cannot make wrong decisions due to a CPU-TFAULT that occurs while the comparison is happening.

When the voter restarts a faulty node, Shield enters into *recovery mode* and starts overwriting the state from a correct node. The copy is incremental: the non-faulty node keeps track of all objects modified during the copying process so that it can still serve new transactions. This incremental state is then pushed to the faulty node for finalizing the copy.

B. Limitations

The main goal of Shield is to provide resiliency to CPU-TFAULTs with an affordable performance penalty. However, its design has some limitations, which are worth mentioning:

First, it reduces the number of cores available for application threads to execute. As an example, in a 48-core machine having 3 nodes of 8 cores each leaves 24 cores to the application for executing. Note that in this configuration a node has 8 cores for running an instance of ObCC. However, given that Shield is a solution for transactional applications, we believe this limitation is not too stringent because the logical contention often prevents the full exploitation of the available hardware. The second drawback is the increased memory consumption. In Shield, each node replicates the shared state, thus increasing the total memory utilized. However, the memory cost is rapidly decreasing and to protect data from corruption, more than one isolated copy of the shared data is needed. The third drawback is the energy consumption. Replication increases consumption, but it also allows error detection and recovery. Effective error detection and recovery mechanisms have often a negative impact on energy because they entail redundancy. Reducing energy's overhead is still an open issue which we plan to address in the future.

V. Network Layer

We design a protocol for establishing a total order of messages (i.e., transaction requests) issued by application threads in the presence of CPU-TFAULTs, and we name the component that is responsible for implementing this protocol as the *network layer*. It provides the first (tentative) delivery of a message in one communication step after its broadcast, and the total order in another step (two in total). As stated in Section III, our total order protocol relies on a monotonically increasing clock, called *clock-service*. We also recall that our total order protocol assumes a reliable and FIFO communication infrastructure. In other words, if a node N_a sends two messages m_1 and m_2 to a node N_b in that order, N_b delivers m_2 only after m_1. No message can be lost.

The pseudo code of the algorithm is reported in Figures 1 and 2. In the following we relate the protocol description to the pseudo code by specifying the line number (e.g., Line x.y means Figure x at line number y). Each application thread sends its requests (`tx_request`) to all the nodes [Line 1.7-1.8] and sends associated acknowledgment-requests (`ack_request`) to other application threads [Line 1.9-1.10]. According to our programming model, a `tx_request` message contains: the application thread ID; the name of the transaction to execute along with its parameters (if any); and the current timestamp. An `ack_request` message contains the ID and the current timestamp only.

When a node receives a new `tx_request` message, it immediately triggers the tentative delivery for that message using the message timestamp as tn-order [Line 2.6]. Since messages cannot be lost in our architecture, nodes do not reply with an *ack*. However, a node N_a receiving a message m_x cannot know whether m_x can be delivered or if there is another message m_y that has been sent before m_x and thus it is currently in transit. Note that m_y can only be sent by an application thread different from the one that sent m_x because, otherwise, due to the FIFO channel, m_y would have been delivered before m_x.

```
1 while(true) {
2   bool ack_requested = false;
3   while(app_queue.dequeue())
4     ack_requested = true;
5   if (!app_requests.empty()) {
6     ack_requested = false;
7     for (i=0; i < node_count; i++)
8       send_tx_request(node[i],app_requests.dequeue());
9     for (i=0; i < application_count; i++)
10      if (i != id) send_ack_request(app[i]);
11  }
12  if (ack_requested)
13    for (i=0; i < node_count; i++)
14      send_ack_msg(node[i]);
15 }
```

Fig. 1. Network Layer - Application thread side.

```
1while (true) {
2  msg = receive_msg();
3  max_timestamp_seen[msg.source] = msg.timestamp;
4  if (msg.data == DATA_MSG) {
5    msgs_queue[msg.source].enqueue(msg);
6    tn_deliver_msg(msg);
7  }
8  found = true;
9  while (found) {
10   min = INFINITY;    min_index = 0;
11   for (i=0; i < application_count; i++)
12     if (!msgs_queue[i].empty())
13       if (msgs_queue[i].top().timestamp < min) {
14         min = msgs_queue[i].top().timestamp;
15         min_index = i;
16       }
17   if (min == INFINITY) {
18     found = false;    break;
19   } else
20     for (i=0; i < application_count; i++)
21       if (max_timestamp_seen[i] < min && i != min_index)
              {
22         found = false;    break;
23       }
24   if (found) {
25     head_msg = msgs_queue[min_index].dequeue();
26     deliver_msg(head_msg);
27   }
28 }
29}
```

Fig. 2. Network Layer - Node side.

For this reason, once an application thread Th receives a `ack_request` message, it has to acknowledge all nodes by sending the current timestamp ts. This message has no payload because it is just needed for letting nodes know that there cannot be any other message from Th with a timestamp lesser than ts. According to that, before deciding to deliver a (totally ordered) message m_x with timestamp ts_x, a node waits until all other application threads have sent a message with higher timestamp. Exploiting the FIFO channel, if a node receives a notification from all other application threads with a timestamp greater than ts_x, it means that all previous sent messages have already been delivered; otherwise, it waits.

At this stage, each node simply selects the message with the minimum timestamp [Line 2.10-2.23] and triggers the delivery for that message [Line 2.24-2.27]. This rule is deterministic and guarantees that messages are ordered equally by all nodes.

Note that our network layer does not let nodes exchange messages; only application threads interact with nodes. This way, a CPU-TFAULT cannot be propagated across nodes.

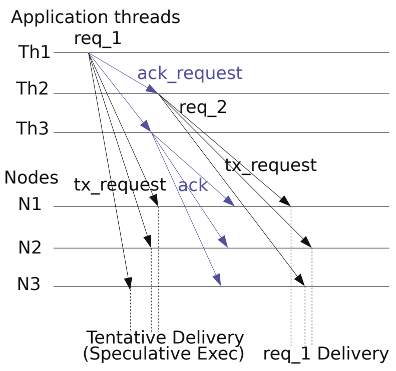


Fig. 3. An example of our total order protocol.

In order to reduce the number of messages sent, we further optimize the notification sent by application threads after receiving a new `ack_request` message from another application thread as follow. Since nodes need only the timestamp of each application thread, the total number of messages sent can be reduced by merging this notification with the broadcast of a new `tx_request` message (if any). In other words, when an application thread receives an `ack_request` message, if it has a new `tx_request` message to broadcast, it does not send the timestamp and then perform the broadcast, but rather, it directly broadcasts [Line 1.5-1.6]. When a node, which is waiting for the notification from that specific application thread, receives a new `tx_request` message broadcast from that thread, it extracts the timestamp associated with the new message and considers it as the notification [Line 2.3]. This optimization reduces the delay for delivering a message.

In Figure 3 we show an example of how the protocol works. Here we have three application threads and three nodes (f=1).

1) Th1 wants to execute a transaction and sends: *i)* `tx_request` (`req_1`) to all nodes; and *ii)* `ack_request` to the other application threads.
2) As soon as a node receives the request, it tentatively delivers the message.
3) Other application threads (i.e., Th2 and Th3) response to the `ack_request` by sending their timestamps. Th3 has no `tx_request` to perform, and hence it just sends its timestamp to all nodes. Th2 already has a `tx_request` (`req_2`) that needs to be sent. Thus, it sends (`req_2`) to all nodes, which already includes the new timestamp.
4) When a node receives `req_1`, `req_2`, and the `ack` from Th3, it can now (non-tentatively) deliver `req_1` which

has the minimum timestamp.

A. *Tolerating* CPU-TFAULT*s*

If a CPU-TFAULT affects the clock-service, Shield will detect that error and resynchronize the clocks again. A bit flip can indeed corrupt the clock value, but anyway the new value will continue to increase after that error as before, since the clock value is always increasing. We can identify the following three possible erroneous cases:

- The first case is a large shift to the past or future of the clock value. This issue is easily detected by comparing the request timestamp with the last timestamp received by the same application thread.
- The second case is a small shift to the past or future. In this case, the protocol continues to order requests as before, but some acknowledgments will be dropped. As long as all application threads are sending requests, the shift will not be detected and the system will continue to order requests uninterruptedly. When requests stop, some of the last requests will likely get stuck in the node queues. A timeout is used to detect this case. In our prototype, the timeout is set to the maximum network delay expected.
- The third case happens when a fault occurs while copying the clock value to the request message. Here one request will be sent with a different timestamp to one node. This error is detected by a timeout in the voter (see Section VII).

VI. Node Concurrency Control

The goal of our concurrency control, called ObCC, is to process transactions once they are tentatively delivered and according to their tn-order. Once the total order for that message is established, then the tn-order is matched with the total order. If those two orders are the same, then the transaction can be committed; otherwise, it has to re-execute. An independent instance of ObCC runs on each node and commits transactions in the part of the memory reserved for that node. In order to reduce the instrumentation overhead, which could hamper the effectiveness of the speculative execution, ObCC provides the highest priority to the next-to-commit transaction (also called the *committer* transaction), which corresponds to the oldest transaction already totally ordered.

Transactions are activated as soon as they are tentatively delivered (*tn-del* hereafter) and their timestamp defines the tn-order (speculative execution order). The conflict of two tn-del transactions is solved by relying on their tn-orders. Consider two transactions T_1 and T_2. Let their tn-order be T_1 followed by T_2. When a conflict occurs, T_2 is aborted and restarted, thus allowing T_2 to access data written by T_1. Non-conflicting tn-del transactions are processed without incurring any aborts because their processing order is equivalent to any other order.

The protocol makes use of the following (major) meta-data: transactional read and write operations are locally buffered in private structures called *read-set* and *write-set*, respectively. The timestamp taken from the clock-service, called `currentTimestamp`, is used to detect changes on read objects. `Orecs` represents the write-lock table, where an entry includes the lock status, the super-lock status, the lock owner, and the version of the last write.

```
1 word txRead(address, tx) {
2   wLock = Orecs.get(address);
3   if (wLock.owner == tx)
4     return writeSet.get(address);
5   do {
6       while(wLock.isSuperLocked || (wLock.isLocked &&
            wLock.owner (precedes) tx))
7           waitForUnlock();
8       v1 = wLock.version;
9       value = memory[address];
10      v2 = wLock.version;
11  } while(v1 != v2);
12  if (v1 > tx.readTS)
13    validate(tx); //or abort if validation failed
14  readSet.add(address);
15  return value;
16 }
17 word committer_txRead(address, tx) {
18     readSet.add(address);
19     return memory[address];
20 }
```

Fig. 4. Read procedure.

ObCC uses write-locks that are acquired at encounter time before performing the actual write operation. A transaction T_w writing object X must acquire the write-lock on X [Line 6.13]. If X is locked by another transaction T_k that is older than T_w (according to either the tn-order or the total order, if defined), then T_w waits until T_k commits [Line 6.8-6.10]. If T_k follows T_w, then T_k is aborted [Line 6.12]. Exploiting this mechanism, conflicting transactions are serialized according to the tn-order, until the total order is established. After that, the total order will be considered as the reference order.

When a transaction T_r reads an object Y, locked by a transaction T_j and T_j precedes T_r, then T_r waits until that transaction finishes [Line 5.6-5.7]. In the opposite case, T_r ignores the lock and reads the object because it is serialized before T_j, therefore T_j's written object is not visible to T_r. In order to guarantee that no transaction is writing the object Y during a read operation on it, Y's version is checked before and after the read [Line 5.8-5.11].

When the read is complete, Y's version is contrasted with T_r's last read-timestamp `readTS`. If it is different, then other transactions in the system have been committed after T_r began its execution. Therefore, ObCC validates T_r's entire read-set to be sure that all the objects accessed are still consistent [Line 5.12-5.13]. If the validation fails, T_r is aborted and restarted [Line 8.15]. Validation of a transaction relies on objects' versions. The procedure compares the versions of all objects in the read-set with the transaction's `readTS` [Line 8.12-8.14]. If validation succeeds, i.e., all the object versions are smaller than transaction's `readTS`, then it is advanced to the current system timestamp [Line 8.11, 8.17]. This minimizes the number of invocations of the validation procedure.

Only one thread is allowed to commit at a time, because ObCC must enforce the total order defined by the network layer as the commit order. No concurrency is allowed at this stage. Say three transactions are tn-del in the order $\{T_1, T_2, T_3\}$ and the total order is $\{T_3, T_1, T_2\}$. The transaction that receives the permission to commit first is therefore T_3, even though it is the last according to the tn-order. Given the mismatch between T_3's tn-order an total order, its accessed objects could be invalid and its written objects could be meaningless. As a result, T_3 must validate its execution before starting the commit phase. Similarly, T_1 and T_2 have to accomplish the same validation procedure before committing.

```
1 void txWrite(address, value, tx) {
2   wLock = Orecs.get(address);
3   if (wLock.owner == tx) {
4     writeSet.update(address, value);     return;
5   }
6   do {
7     if (wLock.isLocked)
8       if(wLock.isSuperLocked || wLock.owner(precedes)tx)
          {
9         waitForUnlock();
10        validate();
11      } else
12        abort(wLock.owner);
13  } while( CAS(wLock.lock, UNLOCKED, LOCKED));
14  wLock.owner = tx;
15  writeSet.add(address, value);
16 }
17 void committer_txWrite(address, value, tx) {
18   wLock = Orecs.get(address);
19   if (wLock.owner != tx) {
20     abort(wLock.owner);
21     wLock.superLock = LOCKED;
22     acquiredLocks.add(address);
23     wLock.owner = tx;
24   }
25   memory[address] = value;
26   writeSet.add(address, value);
27 }
```

Fig. 5. Write procedure.

```
1 void transitionToCommitter(tx) {
2   validate();
3   for (each address in writeSet)
4     acquiredLocks.add(address);
5   for (each address in writeSet)
6     memory[address] = writeSet.get(address);
7   //use committer version of txRead and txWrite functions
8   tx.committer = true;
9 }
10 void validate(tx) {
11   tmp = currentTimestamp;
12   for (each address in readSet) {
13     wLock = Orecs.get(address);
14     if (wLock.version >= tx.readTS && wLock.owner != tx)
15       abort(tx);
16   }
17   tx.readTS = tmp;
18 }
```

Fig. 6. Validation and transition to committer procedure.

When the final order for a transaction is defined, and it corresponds to the next transaction to commit, that transaction has the highest priority in the system and must be committed fast because the application is waiting for its reply. ObCC detects when a speculative transaction becomes non-speculative by simply checking whether its final order has been defined. If the final order corresponds to the next transaction to commit, that transaction's priority is made the highest. We call this execution status: *committer mode*. The main advantage of processing a transaction in the committer mode is that the transaction executes with very low instrumentation, thus it writes directly to the memory [Line 6.25] and it reads without instrumentation [5.19]. If a thread becomes the committer after executing all the transactional operations, it validates its read-set and commits its written objects [Line 8.1-8.9]. After that, it updates the versions of all the locks and releases them. We recall that, in order to provide the response to the application thread, transactions' outcomes should be compared by the voter (see Section VII). Thus, even though a thread is in committer mode, it still has to log its read and written objects into its read-set and write-set [Line 5.18, 6.26].

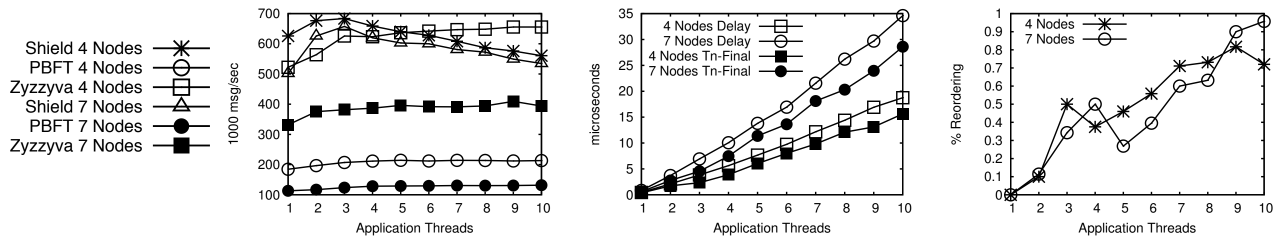
If a thread is promoted as committer when the transaction is still executing, the thread validates the current status of its read-set, then commits its written objects and keeps executing as commiter [Line 8.1-8.9]. For each new write operation, the committer thread acquires a new type of lock, called *super-lock* [Line 6.19-6.24]. This lock is implemented such that no other threads can compete with it. Therefore, no CAS (Compare-And Swap) or atomic operation is required, allowing the committer thread to proceed without blocking its execution on any synchronization point. Clearly, when a super-lock is acquired on an object, no other transactions are allowed to write on that object. Determinism is guaranteed by committing all transactions in the same order on all nodes. However, non-deterministic operations like `random` and `time` are not allowed during the transaction's execution. When needed, the application sends these values along with a transaction request, so that all replicas can use the same non-deterministic value.

There is only one committer thread at a time that executes with the highest priority and following the total order already established. No concurrency is allowed on the commit process and transactions are always validated before entering the committer mode. Therefore, it is straightforward to prove that ObCC guarantees serializability [21].

VII. VOTER

The voter component consists of $2f+1$ voter threads so that even if a CPU-TFAULT occurs during the verification process, no wrong decision can be made. Each thread independently compares the outcomes of the next-to-commit transaction, according to the total order, by matching the gathered read-sets and write-sets from all nodes. When an error is detected (i.e., there is no matching), a voter thread sends to the faulty node a recovering signal. A faulty node starts the recovery process only upon receiving $f+1$ recovery requests, which guarantees that the error actually happened on the node and not on the voter thread. Following the same policy, each voter thread compares the decisions of all other voter threads to confirm its decision matches the majority. Finally, one non-faulty voter thread sends the result to the application thread that originated the request.

When the first verification request is received from a node, a voter thread starts a timer that expires after a certain pre-configured time. This timer is used to detect faults that cause a node to delay or miss sending the verification request with the transaction's outcome. The timer is also used to detect faults of other voter threads that delay the accomplishment of the verification process or are crashed.



(a) Throughput (b) Latency (c) Reordering

Fig. 7. Performance of the network layer with Tilera. Figure 7(c) reports the reordering % between tn-del and final-del.

VIII. Evaluation

Shield is developed in C++. Our test-bed is composed of a hardware message-passing-based architecture, which is the 36-core board of the Tilera TILE-Gx family [7]. Each core is a full-featured 64-bit processor (1.0 GHz), with two levels of caches, 8 GB DDR3 memory, and has a non-blocking mesh that connects cores to the Tilera 2D-interconnection system.

We integrated ObCC into the RSTM library [22]. RSTM uses platform-specific assembly instructions. Therefore, we ported its original implementation to our test-bed. As a result, our new version of RSTM is compliant with all the platforms already supported by RSTM and the Tilera TILE-Gx.

As competitors, we implemented two well-known BFT systems: PBFT [11] and Zyzzyva [23]. PBFT is a state-of-the-art BFT solution that uses a three-phase protocol to deliver totally ordered messages. Zyzzyva is a client-centric protocol that uses only two communication steps when the system has no fault. Both PBFT and Zyzzyva use $3f + 1$ nodes to tolerate f faults. Thus, we selected 4 and 7 nodes in our experiments although Shield requires only $2f + 1$ nodes to tolerate the same number of faults (i.e., 1 and 2 respectively). Our implementation of PBFT and Zyzzyva does not involve message authentication. As for non-fault tolerance protocols, we included TL2 [24], NOrec [25], and SwissTM [26] in our comparison, because each of them exposes different design principles (i.e., NOrec allows a single committer at-a-time; TL2 leverages fine-grain locking; SwissTM is close to the ObCC design as it uses read/write locks and it is Orec-based – i.e., for each shared object, a memory space is defined where meta-data are stored). It is worth noting that those protocols commit transactions in any order while ObCC is forced to commit transactions according to the total order. This requirement introduces an additional overhead for ObCC.

As applications, we evaluate Shield using two well-known benchmarks for transactional systems: TPC-C [8] and Vacation from the STAMP suite [9]; and two micro-benchmarks: the List data structure and Bank, the monetary application. These benchmarks have different characteristics: Bank transactions are very short; TPC-C involves more computation, resulting in longer execution time; Vacation represents a real-world workload by emulating a travel reservation system using an in-memory database; List operations pay the cost of traversing the data structure, increasing the read-set size and the execution time. Each reported data-point is the average of 5 repeated tests after having warmed up the system.

Shield allows multiple threads to interleave their executions on a core because, if a CPU-TFAULT happens on an executing thread, the OS context switch will save the execution state and restore the one of the new thread, thus a CPU-TFAULT cannot be propagated due to a context switch. However, threads of different nodes belong to different OS processes, thus they do not inherently share any memory area. In Shield, each node reserves one core for running its own instance of the network layer. A dispatcher (per node) is used for managing network messages and triggering ObCC upon the receipt of tentative and totally ordered deliveries.

A. Network Layer

In order to evaluate the performance of the network layer without transactional workloads, we conducted an experimental study by varying the number of application threads issuing requests to be totally ordered, and fixing the number of nodes to $\{4, 7\}$. In this experimental study, dispatchers do not activate ObCC, but they only log network messages' meta-data for collecting statistics. Shield's network layer implementation does not batch messages for reducing transaction latency. We measure the throughput (i.e., number of messages totally ordered per second); the average latency between the broadcast and the delivery of a message, as well as the latency between the tentative and final delivery; and the probability of mismatch between the tentative and final order.

Figure 7(a) shows the network throughput. Two factors affect the network throughput: the total number of messages exchanged per request, and the number of communication steps involved for reaching the total order. In PBFT and Zyzzyva, the number of messages varies according to only the number of nodes. Rather, in Shield it depends on the following three factors: the number of nodes, the number of application threads, and the system load (due to the optimization where acks can be piggybacked with new requests). For instance, PBFT requires a total of 28 and 91 network messages per application request when 4 and 7 nodes are deployed, respectively. On the other hand, Shield takes only one communication step under high load (because an application thread does not need to send an acknowledgment when it has a new request ready to broadcast) and therefore the total number of messages per request is in the order of the sum of the number of nodes and the number of application threads.

For that reason we observe the low throughput of PBFT compared to others. Zyzzyva is slightly better than Shield when 4 nodes and a high number of application threads are deployed. However, the additional communication steps have a negative impact using 7 nodes. Shield performance is slightly affected when increasing the number of nodes. With 4 nodes, Shield is similar to Zyzzyva in the range of +20% to -14%. At 7 nodes, Shield is on average 52% better than Zyzzyva. Compared to PBFT, Shield is up to 3.6× better.

Figure 7(b) reports the average time between the broadcast of a request and its delivery ("Delay"). The delay clearly increases along with the number of application threads but it is worth noting that it is still lower than 35 microseconds. The figure also includes the average time between a tentative and non-tentative delivery for a request. This time represents, on average, 76% of the total delay for ordering a request, and it can be exploited by ObCC to speculate while the total order is determined (given that the processing is in-memory). Figure 7(c) shows that in Shield, the speculative execution of ObCC is effective because for the 99% of the cases, there is no mismatch between the tentative order and the total order.

B. System Performance

Here we show the performance of the whole system deployed. We experimented it by using 4 and 7 nodes, and we varied the number of threads forming the ObCC concurrency control (note that those threads are not application threads). We selected this configuration because, this way, we can show the impact of the parallel transaction execution (despite the in-order commit) and its scalability while increasing the ObCC worker threads. Note that PBFT and Zyzzyva process transactions serially by using one thread. In the following experiments, the number of application threads is fixed at 3 and they are configured as open-loop where requests are injected repeatedly with a think-time in between them. The latter has been configured by selecting the one providing the best performance close to the system's saturation.

In Figure 8(a) plots performance using the TPC-C benchmark. The benchmark is configured using the standard percentage of transactions profile as suggested by the original specification, with 100 warehouses available in the system. In TPC-C, transactions are complex and long, thus the load of the transactional computation overcomes any possible bottleneck introduced by the network layer. As a consequence, Shield is able to perform very close, 9% and 22.5%, to the non-fault-tolerant protocols, SwissTM and TL2, respectively. NOrec's behavior on this benchmark is different from others as it does not use Orecs, but, at a large number of threads, it is similar to TL2. The parallelism of ObCC allows Shield to outperform Zyzzyva and PBFT by 1.14× and 1.19× (on average), respectively.

Results with the List benchmark are in Figure 8(b). We configured the list with 256 items, and each transaction selects an operation to execute (i.e., read, insert, and delete) using a uniform distribution. Contention in list operations is high due to traversing all elements up to the required one. Thus, the read-set is large, which increases the conflict rate when a node in the read-set is modified by another transaction. As a general assessment, the results indicate the same trend as that of TPC-C for Shield and SwissTM. The performance of TL2 and NOrec is better as their concurrency control schemes do not use encounter time write locks, which results in a good design choice under the tested workload. Here, the gap between SwissTM and Shield is about 48% and Shield outperforms BFT competitors by as much as 87%.

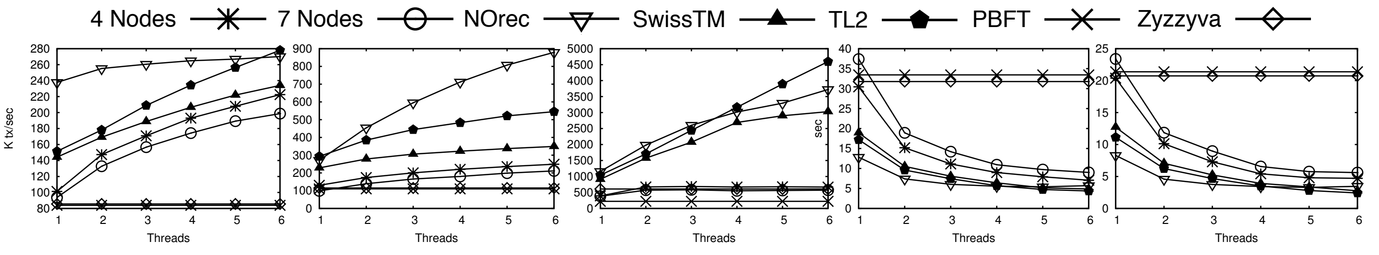
Figure 8(c) shows the Bank benchmark's throughput. Shield's performance is very close to the network layer's throughput. That is because Bank transactions are very short (just a few memory operations) and the speculative processing, that is done before issuing the total order, allows the transaction execution to significantly overlap with the network ordering process. Here the network layer represents the performance bottleneck because the real computation is limited, and thus non-fault-tolerant competitors can achieve better performance than Shield (e.g., 2.4×) because they do not wait for the delivery of a request to start its processing.

Figure 8(d) shows results for the Vacation/STAMP benchmark with high contention configurations. Note that in this experiment, we plot the execution time (in contrast to all previous benchmarks, which plot the throughput), thus, lower is better. Vacation exposes long transaction execution time because it emulates an online travel reservation system, with application threads performing three types of transactions (reservations, cancellations, and updates) that interact with an in-memory database. Starting from 3 ObCC threads, the differences between Shield and SwissTM, NOrec, and TL2 are 44%, 54%, and 58% respectively. Shield is 2× and 2.2× faster on average than Zyzzyva and PBFT respectively. Figure 8(e) shows results using Vacation configured with low contention. At low contention, the number of conflicting transaction decreases and more transactions can run concurrently. As a result, Shield performs slightly worse than before.

Summarizing, the general trend observed in these results is that Shield does not significantly degrade system performance with respect to standalone (i.e., non-fault-tolerant) execution, especially in non-trivial benchmarks (e.g., TPC-C, Vacation). Shield overhead is in the range from 9% to 60%. Shield concurrent execution allows it to get much better performance compared to non-concurrent BFT systems. Shield is 1.54× on average better than non-concurrent BFT systems.

IX. Related Work

In the fault-tolerance literature (from databases to distributed systems), many proposals focused on increasing concurrent execution of requests to improve performance. The key idea is to find independent requests that can run in parallel while forcing dependent requests to run sequentially with the same order in all nodes. In [27], the primary replica executes transactions concurrently. Then a scheduler uses information from primary execution of transactions to drive secondary replicas' execution. This approach is applicable to strict two-phase locking concurrency controls.



(a) TPC-C (b) List (c) Bank (d) Vacation/High Contention (e) Vacation/Low Contention

Fig. 8. Performance of Shield with Tilera. Figures 8(c), 8(a), 8(b) show the throughput; Figures 8(d), 8(e) show the application completion time.

There are also the following proposals for centralized systems. In [12], the system is split using isolated virtual machines which represent servers as in the distributed system. Replicant [28] involved developers by adding determinism hints. Its relaxed determinism model allows concurrent execution unless developers explicitly added a determinism hint. 1Paxos [17] is a consensus algorithm optimized for running on centralized multicore architectures. Its algorithmic innovations focus on how to effectively exploit the available resources. As the other algorithms derived from Paxos [15], 1Paxos targets the crash of executing processors.

Hardware fault-tolerant architectures are more expensive due to the cost of each added replica, and they are limited in the number of faults they can tolerate. For example, NonStop architecture [29] uses two or three replicas. Each replica executes the same instruction stream independently and a voter compares outputs. Hardware resources (memory, disks) are split between replicas and isolated from each other.

State machine replication is a well known paradigm in transaction processing [4]. The notion of tentative order has been introduced in [20] and further exploited in [30], [31].

X. Conclusion

Shield is specialized in solving CPU-TFAULTs, thus it does not suffer from the overhead of general BFT solutions when deployed in centralized (multicore) architectures. Our results on hardware message-passing-based board confirm the claim.

Acknowledgment

Authors would like to thank Prof. Binoy Ravindran for his feedback in the first stage of this work. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367.

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, 2005.

[3] M. Mohamedin, R. Palmieri, and B. Ravindran, "On preserving data integrity of transactional applications on multicore architectures," in *ICDCS*, 2015, pp. 764–765.

[4] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[5] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *EMSOFT*, 2012, pp. 83–92.

[6] G. Yalcin, O. Unsal, I. Hur, A. Cristal, and M. Valero, "FaulTM: Fault-Tolerance Using Hardware Transactional Memory," in *Pespma*, 2010.

[7] Tilera Corporation, *TILE-Gx Processor Family*, http://www.tilera.com.

[8] T. Council, "TPC-C benchmark," 2010.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08*, pp. 35–46.

[10] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, 1982.

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999, pp. 173–186.

[12] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," ser. ATC, 2008, pp. 287–292.

[13] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," ser. SOSP, 2003, pp. 253–267.

[14] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek, "Delta execution for software reliability," ser. HotDep, 2007.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst. '98*.

[16] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," in *OSDI*, 2008, pp. 369–384.

[17] T. David, R. Guerraoui, and M. Yabandeh, "Consensus inside," in *Middleware*, 2014, pp. 145–156.

[18] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up consensus by chasing fast decisions," in *DSN*, 2017, pp. 49–60.

[19] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *DSN*, 2016, pp. 156–167.

[20] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE TKDE*, vol. 15, no. 4, 2003.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, 1987.

[22] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *TRANSACT*, 2006.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," ser. SOSP, 2007, pp. 45–58.

[24] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC*, 2006, pp. 194–208.

[25] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records," in *PPoPP*, 2010, pp. 67–78.

[26] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *PLDI*, 2009, pp. 155–165.

[27] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," ser. SOSP, 2007, pp. 59–72.

[28] J. Pool, I. S. K. Wong, and D. Lie, "Relaxed determinism: Making redundant execution on multiprocessors practical," ser. HOTOS'07.

[29] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," ser. DSN, 2005, pp. 12–21.

[30] R. Palmieri, F. Quaglia, and P. Romano, "Osare: Opportunistic speculation in actively replicated transactional systems," in *SRDS*, 2011, pp. 59–64.

[31] S. Hirve, R. Palmieri, and B. Ravindran, "Archie: a speculative replicated transactional system," in *Middleware*, 2014, pp. 265–276.