HAFT: Hardware-Assisted Fault Tolerance

Dmitrii Kuvaiskii
TU Dresden
dmitrii.kuvaiskii@tu-dresden.de

Rasha Faqeh
TU Dresden
rasha.faqeh@tu-dresden.de

Pramod Bhatotia
TU Dresden
pramod.bhatotia@tu-dresden.de

Pascal Felber
University of Neuchâtel
pascal.felber@unine.ch

Christof Fetzer
TU Dresden
christof.fetzer@tu-dresden.de

Abstract

Transient hardware faults during the execution of a program can cause data corruptions. We present HAFT, a fault tolerance technique using hardware extensions of commodity CPUs to protect unmodified multithreaded applications against such corruptions. HAFT utilizes instruction-level redundancy for fault detection and hardware transactional memory for fault recovery. We evaluated HAFT with Phoenix and PARSEC benchmarks. The observed normalized runtime is 2×, with 98.9% of the injected data corruptions being detected and 91.2% being corrected. To demonstrate the effectiveness of HAFT, we applied it to real-world case studies including Memcached, Apache, and SQLite.

1. Introduction

Transient faults, or soft errors, in CPUs can cause arbitrary state corruptions during computation. Several studies suggest that transient errors are a pervasive cause of software systems failures [29, 48, 64]. These studies point to a wide range of reasons for such transient faults in CPUs, including manufacturing problems, overheating, dynamic voltage scaling, hardware/software incompatibility, or power supply faults.

These issues are amplified in the new processor architectures that are continuously boosting performance with higher circuit density using ever-shrinking transistor sizes, and are simultaneously achieving higher energy efficiency by operating at lower voltages [16]. These trends negatively affect the reliability of the underlying hardware [66]. Furthermore, the advancements in the 7 nm chip technology with near-threshold computing (dim silicon) will only worsen the reliability of CPUs [65].

The unreliability of CPUs becomes a particularly serious concern for modern online services running in data centers. Given the sheer scale at which these services operate, the transient faults occur at a surprisingly high rate and tend to reappear more frequently after the first occurrence [29, 48, 64]. Anecdotal evidence indicates that a single transient fault in the hardware can lead to process state corruption [3, 20], data loss [9], and in some unfortunate cases, errors can propagate throughout the system causing outage of the entire service [1].

As a result, software systems running in modern data centers are being increasingly adapted to tolerate transient faults. For instance, Mesa [31], a data warehousing system at Google, uses application-specific integrity checks to deal with data corruptions during computation. In fact, many large-scale systems employ ad-hoc mechanisms to detect data corruptions, such as source code assertions, periodic background integrity checks, and message checksums throughout the system [1, 22, 31, 35]. However, these ad-hoc solutions can only protect from errors anticipated by the programmer and may fail to detect arbitrary hardware faults.

Researchers have proposed a series of disciplined *hardening* approaches to protect software systems against transient faults [40, 60, 67, 79, 80]. In particular, these hardening approaches add redundancy at the level of program instructions, threads, or whole processes, and insert periodic comparisons of redundant copies to detect transient faults. While these approaches have been an active area of research for decades, almost all of the existing solutions in this domain target sequential programs only, making them impractical for ubiquitously deployed multithreaded software systems.

To support multithreaded programs, a few hardening systems have been recently proposed [11, 13, 24]. However, all these systems still have at least one of the following limitations: *(i)* they require manual efforts to modify the application, e.g., to annotate the protected code regions, *(ii)* they target restrictive programming models, e.g., assuming only event-based applications, *(iii)* they rely on application-specific checks leveraging the high--level programming languages such as Apache Pig [50], *(iv)* they require operating system support, deterministic multithreading and/or spare cores for redundant execution, *(v)* they provide only fail-stop semantics without providing recovery from faults.

EuroSys '16, April 18-21, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901339

In this paper, we propose a Hardware-Assisted Fault Tolerance (HAFT) technique that overcomes the aforementioned limitations. HAFT applies to unmodified applications on the existing operating systems running on commodity hardware. HAFT targets the general shared-memory multithreaded programming model supporting the full range of synchronization primitives. Moreover, HAFT neither enforces deterministic execution nor requires spare cores, and thereby, it does not limit the available application parallelism, which is crucial for imposing low performance overheads. Finally, HAFT achieves high availability by providing fault detection *as well as* recovery from faults.

HAFT is a compiler-based hardening approach that leverages two techniques: Instruction-Level Redundancy (ILR) [60] for fault detection and Hardware Transactional Memory (HTM) [78] for fault recovery. To achieve fault tolerance, HAFT transforms an application in the following way. First, the instructions of the application are replicated and periodic integrity checks are inserted. The replicated instructions create a separate data flow along the original one, and both flows are efficiently scheduled via instruction-level parallelism of modern CPUs. Next, the whole execution of a program is covered with HTM-based transactions to provide fault recovery. When a fault is detected by ILR, the transaction is automatically rolled back and re-executed. The HTM implementation we employ is best-effort, which renders HAFT's recovery guarantees best-effort as well. Nonetheless, our evaluation shows that clever placement of transactions allows HAFT to achieve high availability even in the presence of frequent faults.

We implemented HAFT as an extension of the LLVM compiler framework to transform unmodified application code. In our evaluation, we applied HAFT to the Phoenix and PARSEC benchmark suites. The fault injection experiments show that the average number of data corruptions decreases from 26.2% to 1.1% and on average, 91.2% of the data corruptions can be corrected. In terms of performance, applications hardened with HAFT run on average 2× slower than native versions. We also applied HAFT to a set of real-world applications including Memcached, Apache, and SQLite. Furthermore, a comparative evaluation revealed that HAFT imposes 30–40% less performance overhead than the state-of-the-art solution for multithreaded programs [11].

2. Background and Related Work

We discuss below existing approaches to fault tolerance and uses of hardware transactional memory for fault recovery.

2.1 Fault Tolerance Approaches

State Machine Replication (SMR). To achieve high availability, some software systems [10, 17, 34] use State Machine Replication (SMR) [63]. These systems typically assume a crash fault model. However, this model does not cover transient faults which might lead to arbitrary state corruptions.

Byzantine Fault Tolerance (BFT) [19] tolerates not only crashes, but also transient hardware faults (and even malicious attacks). Unfortunately, BFT incurs prohibitive overheads because of the overly pessimistic fault model. To decrease the performance overhead of BFT, researchers explored the use of specialized trusted hardware [36, 71], relaxed network assumptions [54, 55], speculative execution of requests [39], and OS support [38]. In contrast, HAFT imposes low overheads by assuming a *more restrictive* fault model: it protects only against hardware non-malicious faults.

To support multithreaded programs, all SMR solutions require some form of deterministic execution. Crane [23] builds on deterministic multithreading [45, 51], Eve [37] speculatively executes requests and falls back to deterministic re-execution upon conflicts, and Rex [30] enforces deterministic replay of the primary's trace on secondary replicas. HAFT supports non-determinism because it requires no replicas, achieving fault tolerance *locally*.

Due to its local fault tolerance, we consider HAFT to be not a substitute for SMR, but rather a complementary approach. Indeed, SMR is usually applied only to the "control path" of distributed software systems, e.g., coordination services such as Chubby [17] and ZooKeeper [34]. HAFT can, in particular, be used to protect the data path, ensuring that the main computation itself is not affected by transient faults.

Local hardening approaches. Due to lack of adoption of BFT [69], researchers actively explored local hardening approaches that protect against data corruptions. These approaches *harden* programs by adding redundancy at the level of program instructions (see §3.2), threads, or processes.

Redundant Multithreading (RMT) [47, 72, 80] spawns an additional, trailing thread for each original thread in a program and redundantly executes it on a spare core. In the same spirit, Process-Level Redundancy (PLR) [24, 67, 79] uses redundant processes instead of threads, with processes-replicas having their own private memory space and synchronizing on system calls. Both of these approaches require spare cores for redundant execution and are thus not suitable for multithreaded programs that tend to occupy all available cores. Moreover, they only support deterministic program executions.

Scalable Error Isolation (SEI) [11, 22], a recently proposed fault detection technique, is the only approach we are aware of that does not require deterministic execution of multithreaded programs. It assumes an event-driven programming model, executing each event handler twice and appending a CRC signature to all output messages. Thereby, SEI guarantees end-to-end protection from data corruptions in a distributed environment. Unfortunately, SEI requires manual effort to adapt existing code bases. HAFT, in contrast, applies to unmodified programs and targets the common shared-memory programming model. Finally, the authors of SEI assume a broader fault model than HAFT, with no bound on the number of corrupted variables per one event handler, and formally prove the correctness of SEI under this model. HAFT provides weaker guarantees with the benefit of better performance (§6.1).

Most of the approaches above only provide fault detection and fail-stop behavior. Coupling them with fault recovery mechanisms [56, 59, 68, 70] is considered a non-trivial task. HAFT, on the other side, seamlessly combines fault detection and fault recovery.

Lock step CPUs. Traditionally, incorrect execution of programs has been detected via lock step CPUs, where two CPUs execute the same application in parallel and synchronize their outputs. Lock step CPUs are still actively used for critical applications in the embedded domain and on mainframes. By its very nature, lock-stepping requires deterministic core behavior and cannot be applied to modern CPUs that have become increasingly more non-deterministic [12]. Moreover, lock step CPUs provide only fault detection, requiring a separate mechanism for recovery. Being a more light-weight technique, HAFT supports automatic recovery and non-determinism both on the application as well as on core level.

2.2 Leveraging HTM for Fault Recovery

Transactional memory was first proposed as a better alternative for traditional lock-based synchronization in concurrent shared-memory applications [33, 46]. However, it also provides strong isolation guarantees and local rollback and can be exploited as a recovery technique [26].

Intel TSX. In this paper, we focus on a recent HTM implementation called Intel Transactional Synchronization Extensions (TSX) [78]. More specifically, we use the Intel Restricted Transactional Memory (RTM) interface.

RTM introduces a set of new instructions to explicitly begin, commit, and abort transactions. Applications can mark the boundaries of transactions using XBEGIN and XEND, explicitly abort them using XABORT, and check if a CPU core is currently executing in a transaction using XTEST.

In Intel TSX [44, 73, 78], transactions utilize the L1 data cache as a local buffer to track their read- and write-sets. An optimized cache coherency protocol is used to detect collisions between concurrent transactions. Read- and write-sets are implemented at the (64-byte) cache line granularity. A cache line that is part of the read-set can be evicted *without* necessarily causing the transaction to abort, while evicting a cache line that is part of the write-set *always* aborts the transaction.

Internally, XBEGIN commands the core to take a snapshot of its register state and to start tracking the changes done by the transaction in the read- and write-sets. If the core detects a conflict with another transaction (or even with non-transactional code), it aborts its transaction. Otherwise, upon execution of XEND, the transaction commits by atomically flushing its write-set to RAM. If the transaction was aborted (either implicitly or explicitly via XABORT), its read- and write-sets are discarded, the registers' state is restored from the snapshot, and the execution jumps to an abort handler specified as argument to XBEGIN. The abort handler is usually implemented to retry a transaction several times before resorting to a fallback path.

Applicability to fault tolerance. Given that Intel TSX is targeted primarily for synchronization, it is not immediately obvious whether it can be also used for fault tolerance. Although

(a) Native	(b) ILR	(c) HAFT
		xbegin
z = **add** x, y	z = **add** x, y	z = **add** x, y
	z2 = **add** x2, y2	z2 = **add** x2, y2
	d = **cmp neq** z, z2	d = **cmp neq** z, z2
	br d, **crash**	**br** d, **xabort**
		xend
ret z	**ret** z	**ret** z

Figure 1: HAFT transforms original code (a) by replicating original instructions with ILR for fault detection (b) and covering the code in transactions with Tx for fault recovery (c). Shaded lines highlight instructions inserted by ILR and Tx.

some research has recently shown promising results when using HTM for recovery [32, 76, 77], the question remains: can commodity-hardware HTM implementations provide efficient and comprehensive support for fault recovery?

In HAFT, the whole application must be wrapped in hardware transactions to support fault recovery. Yet, several design choices of Intel TSX are driven by the assumption that transactions cover only a handful of small critical sections. This limits TSX's applicability for the whole-application fault recovery in the following ways. Firstly, Intel TSX provides no guarantees that a transaction will eventually commit even when applied to sequential code [78]. Secondly, transaction size is limited by the CPU cache size and by the interval between timer interrupts. For example, TSX has the following rough thresholds after which more than 10% of transactions abort: 16 KB for the write set, 1024 KB for the read set, and 1 million CPU cycles (approx. 0.3 ms) [44, 73]. Thirdly, all interrupts/signals (including page faults) and so-called "unfriendly" instructions (x87 floating-point, TLB or EFLAGS manipulation, system calls) force a core to abort any active transactions.

Thus, to guarantee forward progress, HAFT needs a non-transactional fallback path in case transactional execution does not succeed. Consequently, if a fault happens during one of these non-transactional fallbacks, it cannot be recovered. Moreover, a HAFT transaction must be sufficiently small to finish before a timer interrupt happens or the L1 cache overflows. Finally, several factors such as CPU hyper-threading, memory false sharing, and unfriendly instructions also negatively affect HAFT's recovery capabilities.

3. HAFT

HAFT is a compiler-based transformation that consists of two components: ILR for fault detection and Tx for fault recovery. Figure 1 shows an example of HAFT transforming a simple code snippet. ILR is applied first, replicating all instructions except control-flow ones (Figure 1b). To achieve fault detection, ILR inserts a check before returning the result; if two copies of data diverge, then a fault is detected and an error is reported by enforcing program termination. To achieve fault recovery, Tx is applied next, covering the code in transactions and substituting crashes by transaction aborts (Figure 1c). In this case, if a fault is detected at run-time, the current transaction is rolled back and re-executed. HAFT attempts to re-execute aborted transactions for a bounded number of times (three by default in our implementation), after which the code executes non-transactionally until a new transaction begin is encountered. If a fault occurs during such a non-transactional part of code, ILR has no other choice but to terminate the program. Therefore, HAFT provides best-effort fault recovery, falling back to fail-stop semantics in rare cases when the limit of re-executions is exhausted.

3.1 System Model

Before we explain the basic design of HAFT, we present the system model assumed in this work.

Fault model. HAFT protects against single event upsets (SEU), i.e., a corruption of a single CPU register or a single miscomputation in a CPU execution unit that would otherwise lead to Silent Data Corruptions (SDC) [16]. The SEU model covers transient hardware faults due to particle strikes, aging, dynamic voltage scaling, device variability, etc. We assume that at most one SEU fault occurs during one hardware transaction. HAFT can probabilistically protect against bursts of faults as long as duplicated data flows result in differing corrupted state. Due to the choice of ILR for fault detection, HAFT cannot tolerate common-mode failures; however, single uncorrelated bit-flips are considered to be the dominant cause of CPU faults [16].

Additionally, HAFT assumes that RAM and caches are already protected by ECC [60]. This assumption usually holds for data center servers, e.g., our experimental machine has memory ECC support and all cache levels are protected by ECC or parity.

The design of HAFT assumes correct execution of Intel TSX. The TSX transactional state resides in the L1 cache and thus is protected by ECC. However, if XBEGIN, XEND, or XABORT perform an erroneous operation (e.g., not all cache lines are flushed to RAM or rolled back), the program state becomes inconsistent.

Memory model. HAFT relies on the Release Consistency (RC) memory model [28], which requires that all shared memory accesses are done via synchronization primitives. The RC model guarantees correctness for data-race free programs and enables the optimizations on shared memory accesses (§3.3) which would not be feasible under stricter memory models such as sequential consistency [42]. Indeed, a data race would lead to a discrepancy in results under our optimized ILR that in turn would lead to either a transaction abort (if executed inside an HTM transaction) or a program crash (if executed in non-transactional part of code). To allow for the shared memory accesses optimization, we assume data-race free executions.

Synchronization model. Our current implementation supports POSIX threads API and C/C++ atomic synchronization primitives. In fact, HAFT works with any synchronization mechanism that maps directly to LLVM atomic instructions [6]. Thus, even lock-free programming patterns are supported as long as they are explicitly implemented via atomics. Ad-hoc synchronization mechanisms such as user-defined spin locks are not supported, but they are error-prone and not recommended for use [75].

HAFT is not readily applicable to HTM-enabled applications. Our current prototype does not expect TSX instructions in the native program and therefore could break semantics assumed by the programmer. However, in §6.1 we show that HAFT can be efficiently expanded to applications that use lock elision as their main synchronization primitive.

3.2 Basic Design

In the following, we describe the basics of ILR and TX. For simplicity of presentation, we first consider sequential applications. We then show in §3.3 that HAFT's basic design naturally extends to multithreaded programs and we further enhance it with optimizations to improve performance and reliability.

Instruction Level Redundancy (ILR). HAFT utilizes Instruction Level Redundancy (ILR) for fault detection [25, 40, 49, 60]. ILR operates on one copy of the memory state and checks the results of computations before each update to memory. This way, ILR does not increase the memory footprint and allows non-determinism in applications, selective hardening of functions, and interoperability with legacy libraries.

To add redundancy, ILR creates a second, shadow data flow along the master flow, with shadow instructions working on their own registers (see Figure 1b). Note that the shadow instructions are executed in the same thread. Since there are no dependencies between master and shadow instructions, they can execute in parallel, benefiting from the instruction-level parallelism present in all modern CPUs.

The basic version of ILR replicates all instructions except control flow (branches, function calls, returns) and memory-related (loads, stores, atomics) instructions. If a non-replicated instruction returns a value, as in case of loads and function calls, this value is immediately replicated for later use in the shadow data flow using a register-to-register move.

To achieve fault detection, ILR inserts checks on every instruction that updates memory or control flow. Each check compares a master and shadow data copies, reporting an error upon detecting a discrepancy (Figure 1b, lines 4-5). ILR has a few *windows of vulnerability*, i.e., it cannot detect faults occurring in-between the checks and the checked instructions [60].

In the context of this work, the important advantages of ILR are its fine-grained checking and in-thread redundancy. As we utilize HTM for recovery, we are restricted to transactions of small size operating on a single core. The small size of transactions implies that the checks must be inserted as close as possible to the potential sources of transient faults. The single-core requirement implies that the fault detection mechanism must not use additional cores. ILR fulfills both these requirements.

Transactification (TX). In addition to ILR for fault detection, HAFT also employs transactification (TX) to achieve fault recovery. The TX pass of HAFT inserts transaction boundaries in an application so that it always executes inside HTM transactions. The challenge here is to determine correct transaction boundaries. HTM is traditionally used to protect critical sections, with tiny transactions scattered around the code. In that case, the program-

```
int c = 123;                                    ;; Original C code
void foo() { while (c < 1000) c++; }
```

```
1  entry:                                       ;; Basic block 1
2    tx-begin()
3    dup c.init = load c.adr
4  loop:                                        ;; Basic block 2
5    tx-cond-split()
6    dup c = phi [c.init, entry], [c.new, loop]
7    dup c.new = add c, 1
8    dup cnd = cmp eq c.new, 1000
9    tx-counter-inc(7)
10   br cnd, end, loop
11 end:                                         ;; Basic block 3
12   store c.new, c.adr
13   c.tmp = load c.adr2                      ;; ILR check ⌉
14   d = cmp neq c.tmp, c.new2
15   br d, xabort                                            ⌋
16   tx-end()
```

Figure 2: HAFT transactification example: original C code (top) and LLVM IR generated for it (bottom). Lines 3 and 6-8 show original instructions replicated by ILR, lines 12-15 show a check on store inserted by ILR. Shaded lines highlight calls to HTM helper functions inserted by TX.

mer herself assigns transaction boundaries and ensures the optimal transaction size. HAFT, however, is a fully automated technique that transparently covers the whole application with transactions at compile-time. Thus, an algorithm to efficiently put transaction boundaries—a *transactification* algorithm—is required.

To best illustrate the mechanisms underlying the transactification process, consider the simple example shown in Figure 2.[1] It consists of a single function incrementing a global variable within a loop.[2] Here, ILR is first applied on original code: instructions on lines 3 and 6-8 are replicated, and a store instruction is augmented with a check on lines 12-15; for simplicity, we omit the check before a branch on line 10; refer to §3.3 for details. Next, Tx is invoked to insert transaction boundaries.

A simple transactification algorithm would be to insert boundaries *only* at the level of separate functions (lines 2 and 16). But in reality functions can be arbitrarily large and can in turn call other functions, whereas hardware transactions are severely restricted in size as discussed in §2.2. Therefore, transactions are bound to abort under this naïve approach, i.e., the rate of successfully committed transactions would be prohibitively low.

Another extreme is to cover each *basic block* (single entry single exit section of code) in a separate transaction. In this case, since basic blocks usually contain just a handful of instructions, all transactions should eventually commit. In our example, we would have three transactions covering the three basic blocks (lines 1–3, 4–10, and 11–16). However, the second basic block corresponds to the body of the loop that executes several hundreds of times, creating several hundreds of tiny transactions at run-time. Unfortunately, producing that many transactions introduces high performance penalty (see §5.3).

Therefore, to achieve high commit rate and low performance overhead, TX takes a balanced approach and inserts hardware transactions at the granularity of functions *and* loops. The algorithm tries to maximize the size of transactions, while at the same time keeping it less than a predefined *threshold* to avoid capacity aborts and ensure that the majority of transactions can commit successfully. To that end, given that the size of transactions is not always known at compile-time because the number of loop iterations is not always known statically, TX keeps track of the number of instructions executed inside transactions at run-time using per-thread instruction counters.

TX inserts transactions at compile-time by inspecting all functions in the application and applying a transformation pass that adds transaction demarcations at specific locations. It relies upon the following helper functions that embed the low-level HTM instructions necessary for transactional execution:[3] *(i)* tx-begin() starts a new hardware transaction and resets the thread-local counter. If the transaction does not succeed after a number of retries (default is three), the code executes non-transactionally. *(ii)* tx-end() commits the current transaction. *(iii)* tx-cond-split() if the thread-local counter exceeds a predefined threshold, commits the current transaction, starts a new hardware transaction, and resets the counter. *(iv)* tx-counter-inc() increments the thread--local counter by the number of instructions given as parameter.

For each function in an application, TX first inserts a transaction begin at function entry (line 2) and a transaction end before function return (line 16).

After that, loops are transformed. For each loop, TX inserts a conditional statement at the entry point to commit the current transaction and start a new one only when the instruction counter exceeds a predefined threshold (line 5). This optimization yields significant performance gains since the counter check is significantly cheaper than systematically starting a new transaction at each iteration.

The instruction counter is incremented at each loop latch, i.e., at each point where the execution can jump back to the entry point of the loop (line 9). The increment value is computed as the longest path in the loop body leading to the latch, i.e., it corresponds to a worst-case scenario and the counter represents an upper bound of the transaction size. In the example, the increment value of 7 corresponds to 3 original instructions in the loop, 3 shadow instructions added by ILR, and one branch instruction. Note that a fault in the instruction counter is benign: the corrupted counter can force a transaction to prematurely commit or to unexpectedly abort. In either case, the counter will be reset as soon as a new transaction starts.

Using this loop transformation, several loop iterations can be executed at run-time before the threshold is reached and

[1] We use a simplified LLVM IR notation; the `phi` instruction selects a value depending on the predecessor of the current block.

[2] Note that, for the sake of illustration, we have simplified the generated LLVM code and discarded certain compiler optimizations.

[3] The code of these functions consists of just a few instructions that are subsequently inlined by the optimizer for performance reasons.

(a) Unoptimized

```
;; Load (atomic)
d = cmp neq adr, adr2
br d, xabort
val  = load adr
val2 = move val
;; Store (atomic)
d = cmp neq val, val2
br d, xabort
d = cmp neq adr, adr2
br d, xabort
store val, adr
```

(b) Optimized

```
;; Load (race-free)
val  = load adr
val2 = load adr2

;; Store (race-free)
store val, adr
tmp = load adr2
d = cmp neq tmp, val2
br d, xabort
```

Figure 3: Memory accesses in ILR. Unoptimized (a) is used for atomic accesses while optimized (b) is safe for race-free programs. Shaded lines highlight instructions of the original master flow.

a new transaction begins. Thereby, this technique minimizes the number of required hardware transactions. Note that these transformations are applied recursively to nested loops.

Finally, Tx inserts transaction boundaries around function calls. In the general case, Tx does not know which function is called and for how long it executes, therefore it pessimistically ends the current transaction before the call and begins a new one after it.

3.3 Advanced Features and Optimizations

To reduce the performance overhead of HAFT and increase its reliability, we apply a number of optimizations on ILR and Tx.

Shared memory accesses. In basic ILR, each load and store requires expensive checks (Figure 3a). This can yield significant overheads since, in an average application, approximately 10% of instructions are stores and 30% are loads [14, 15]. In other words, around 40% of the original instructions need checks under the naïve ILR interpretation.

To reduce the number of checks, previous research has assumed a very relaxed memory model with two consecutive loads on the same address always returning the same value [60]. This assumption holds for sequential applications but is violated in multithreaded environments. In contrast, our refined ILR distinguishes between different types of memory accesses and applies optimizations only when they are safe.

The key enabler for our optimizations is the RC memory model (see §3.1). Our design assumes data-race free programs, where all accesses to shared memory are protected via locks or done explicitly using atomics. As such, we can separate memory accesses into atomic and regular ones. Atomic operations are not replicated and require (expensive) checks, while regular memory accesses optimize away most checks by relying on (cheaper) memory loads.

This optimization is illustrated in Figure 3b. By replicating regular loads, we can eliminate the checks of load addresses. Indeed, the data-race freedom assumption guarantees that both master and shadow loads read the same value in the error-free case. A

(a) Naïve

```
d = cmp neq cnd, cnd2
br d, xabort
br cnd, trueblk, falseblk
```

(b) Safe

```
 br cnd, strueblk, sfalseblk
strueblk:        ;; Shadow blocks
 br cnd2, trueblk, xabort
sfalseblk:
 br !cnd2, falseblk, xabort
```

Figure 4: Control flow protection in ILR. The naïve approach (a) does not protect the condition while the safe one (b) does.

fault happening during one of the loads will result in a wrong value being read and will propagate further until it is detected at some later point. Since almost all loads are considered regular, this optimization alone leads to up to 40% reduction in overhead (see §5.3). On the contrary, for the rare cases of *atomic loads*, we cannot perform any optimizations and fall back to an expensive address check and a shadow move for each load (Figure 3a, top).

The case of stores is more sophisticated. As atomic stores are considered irreversible externalization events, all checks must be performed before the store (Figure 3a, bottom). The effects of regular stores are, however, thread-local or protected by locks, which enables us to place the check after the store and simplify it with the help of an extra load (Figure 3b, bottom). Performance-wise, the load and check operations are coalesced in an effective `cmp` x86-instruction, and the additional load does not introduce any latency since it utilizes the store-buffer forwarding feature available on modern CPUs.

Control flow protection. ILR protects against the important class of transient faults that affect the status register (EFLAGS in x86) and result in taking incorrect branches. These faults are especially threatening in control flow intensive applications. For example, 20% of data corruptions in one of the benchmarks (*linearreg* in Figure 9, right) are due to such faults.

Since there is no way to replicate the status register, the basic version of ILR checks branch conditions before a branch instruction (Figure 4a). However, if the condition variable cnd becomes faulty in-between the check and the actual branch, the program flow can diverge undetectably and lead to further data corruptions.

Our refined ILR removes an explicit check on the condition and substitutes it with shadow basic blocks that evaluate the shadow condition and signal an error if a mismatch is detected (Figure 4b). The strueblk shadow basic block is taken if the master condition cnd evaluates to true, and therefore the shadow condition cnd2 must also evaluate to true; otherwise an error is signaled. The same reasoning applies to the sfalseblk block, which operates on an inverse shadow condition. The destinations of the original branch are rewired to the shadow blocks and a transient fault in the status register cannot remain undetected.

Note that ILR does not protect against arbitrary control-flow errors, in particular transient faults that set the program counter (PC) register to some invalid value. Saggese et al. [61] show that a random value in the PC virtually never leads to data corruptions, i.e., there is no need to protect the PC.

Fault propagation check. The design of HAFT assumes that a fault happening in a transaction is quickly detected and handled. There is, however, one corner case when the fault can propagate to a subsequent transaction: the compiler can move stores as part of the loop hoisting optimization, as the example in Figure 2 shows. Here the global variable `c` is incremented in a loop. For performance reasons, the compiler has moved the load of the initial value before the loop (line 3) and the store of the final value after the loop (line 12).

In this scenario, a fault corrupting `c` in one transaction may propagate to the next transaction if the fault happens during loop execution. This problematic case arises from the fact that ILR inserts a check on `c` only at the final store (lines 12–15).

To limit the propagation of faults inside such loops, we developed the following optimization, called a *fault propagation check*. ILR analyzes each loop induction variable and, if it is not covered by in-loop checks, adds an explicit check at the loop entry. TX recognizes these additional checks and moves them inside the conditional transaction split, such that the checks are performed directly before committing the previous transaction. In this case, if a fault corrupts a variable, it will be detected by the newly added checks and the transaction will abort without the fault propagating further.

Local function calls. As described in §3.2, TX inserts unconditional transaction begins and ends at each function entry, function call, and function return. This is a very conservative stance which does not rely on any knowledge of the relationship between different program functions. We notice, however, that most program functions are local, i.e., they are always called from other HAFTed program functions. At the same time, there are some functions that are called from third-party libraries, e.g., `main`.

TX exploits this distinction between local and externally called functions by performing the following optimization. If a function is marked as local, calls to this function are surrounded merely by a counter increment and a follow-up conditional transaction split. Similarly, a local function uses a conditional transaction split at its entry and a counter increment upon return. With this caller-callee interaction in place, TX eliminates two unnecessary transactions at each function call. In our current implementation, the developer is required to provide a black-list of externally called functions for this optimization.

Lock elision. HAFT also supports an original approach for lock elision, which consists in substituting (*eliding*) locks with hardware transactions to gain better performance [57]. The key observation is that at run-time locks are often unnecessary because many critical sections do not overlap in time and could execute safely without locks. In this case, speculative execution of a critical section in a transaction is faster than lock-based execution.

The lock elision optimization in HAFT relies on the fact that hardware transactions can be used for fault recovery *and* lock elision at the same time. We implement this optimization in the following way. Whenever HAFT detects a call to a lock function (acquire or release), it does not surround it with a transaction end and begin, but instead it calls a corresponding wrapper. The wrapper checks if a thread already executes in a transaction. If so, the critical section is executed under the protection of the active transaction without acquiring the lock. Otherwise, HAFT falls back to the original conservative locking scheme. We found this optimization to be particularly helpful in case of Memcached, and we investigate its gains in §6.1.

4. Implementation

We implemented HAFT as a LLVM-based compiler framework [43] that takes unmodified source code of an application and produces a HAFTed executable (§4.1). Additionally, we implemented a software-based fault injection framework compatible with Intel TSX (§4.2).

4.1 HAFT Compiler Framework

Tool chain. We developed HAFT based on LLVM 3.7.0. In particular, we implemented HAFT as two independent LLVM passes: ILR to add fault detection capabilities (∼830 LOC) and TX to add fault recovery (∼540 LOC). Both passes abstract away the underlying details of the architecture; the architecture-specific functionality is extracted in separate LLVM IR files that are queried during compilation.

Overall, the build process proceeds as follows. First, all source files are compiled separately and linked to produce a single LLVM bitcode file [43]. Thereafter, all regular LLVM compiler optimizations are performed on the bitcode representation. We then take the optimized bitcode and pass it through the two implemented compiler passes, namely, ILR followed by TX. Finally, the target machine code is generated. Note that we neither impose restrictions on the traditional compiler optimizations, nor do we require changes to the build parameters.

ILR pass. For the implementation of the ILR compiler pass, we had to modify the LLVM CodeGen module. In particular, since ILR introduces redundant shadow registers and shadow instructions, the LLVM compiler is free to optimize away these shadow copies. To prevent LLVM from doing it, we decouple the master and shadow data flows by introducing CodeGen-level `move` pseudo-instructions and corresponding LLVM intrinsics. These instructions and intrinsics are opaque to all LLVM optimization passes and are replaced by real x86 register moves only at the very last stage of code generation.

Furthermore, the LLVM optimizer can also remove shadow loop induction variables in cases when the initial (constant) value for the variable is known. We prevent this optimization by moving the initial value to a global volatile variable and reading it before the loop body. This trick has negligible performance impact since the initial value is loaded only once before the loop.

For the shared memory access optimization of ILR described in §3.3, we insert a volatile shadow load to prevent the compiler from optimizing it away or moving it around other memory-related operations.

TX pass. The TX pass follows closely the description in §3.2. We introduce thread-local instruction counters and four helper functions, as well as wrappers for the acquire and release func-

FI result	Description	System
Hang	Program became unresponsive	Crashed
OS-detected	OS terminated program	
ILR-detected	ILR detected, Tx did not recover	
HAFT-corrected	ILR detected, Tx recovered	Correct
Masked	Fault did not affect output	
SDC	Silent data corruption in output	Corrupted

Table 1: Classification of fault injection results.

tions from the lock elision optimization in §3.3, in a separate LLVM IR file. The Tx pass queries this file during compilation. This way, we can abstract the Tx pass from the underlying hardware and pthreads implementation.

The threshold for transaction sizes (§3.2) and a black-list of non-local functions (§3.3) are specified using additional LLVM compiler flags.

Collaboration of ILR and Tx. The fault propagation check described in §3.3 requires a tight collaboration between otherwise independent ILR and Tx. To achieve this, ILR adds checks with associated LLVM metadata in the loop. Tx recognizes these checks and moves them in a conditional transaction split right before the previous transaction's commit. The fault propagation check currently works only on innermost loops. Only induction variables from the loop header that are not checked in the loop body are covered by this check.

Both ILR and Tx introduce some basic peephole optimizations, e.g., ILR removes checks that immediately follow a creation of a shadow copy and Tx removes pairs of transaction starts followed immediately by transaction ends.

Libraries support. HAFT can transform only the source code available during compilation. This becomes a problem for applications that rely heavily on external libraries such as libc or libstd++. In such case, these unprotected libraries constitute a significant part of runtime execution and faults happening in their code go undetected. To increase fault coverage for C/C++ applications, we applied HAFT to a part of the libc library and link it to the final executable. We use the musl library [8] with assembly support disabled as reference implementation. We opted not to include dynamic memory allocation, I/O, OS, and pthreads-related functions for our prototype. Firstly, they account for a small fraction of runtime (less than 5%) for most programs, and secondly, they use system calls and unfriendly instructions prohibited in hardware transactions. Notice that most previous systems [25, 59, 60] did not apply their hardening techniques to external libraries, which impedes a direct comparison.

Limitations. Our HAFT prototype does not transform inline assembly code nor assembly functions: LLVM treats assembly as black-box function calls with no additional knowledge of their behavior. Furthermore, our prototype does not protect the C++ exception handling mechanism which requires a tight collaboration of LLVM IR and libstd++.

4.2 HAFT Fault Injection Framework

Fault injection tool. For conducting fault injection experiments of HAFT, we need a software-based fault injection tool that

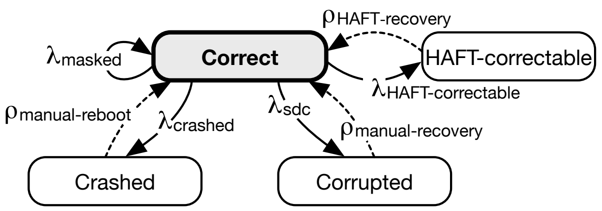
$\rho_{\text{HAFT-recovery}}$
λ_{masked}
Correct
HAFT-correctable
$\lambda_{\text{HAFT-correctable}}$
$\rho_{\text{manual-reboot}}$
λ_{crashed}
λ_{sdc}
$\rho_{\text{manual-recovery}}$
Crashed
Corrupted

Figure 5: HAFT probabilistic model. System transits from correct state to other states at predefined fault rates λ and returns back to correct state at predefined recovery rates ρ.

works with Intel TSX. As other tools [11, 62, 74] do not have such support, we developed our own binary-level fault injector (∼320 LOC).

Our fault injector is based on the Intel SDE emulator [4], which allows us to attach the GDB debugger to an emulated program. We leveraged this feature to design a simple GDB script-based fault injection tool. Intel SDE *emulates* all TSX instructions and thus enables us to perform fault injections on machines that do not have hardware support for TSX. It has an additional benefit that attaching GDB during a hardware transaction does not lead to a transaction abort.

The fault injection experiments proceed in two steps. In the first preparatory step, a reference execution trace of a tested program is generated using Intel SDE's debugtrace tool. This trace contains all the instructions executed by the program and all the registers updated by these instructions. Additionally, the program is run without any fault injections to produce a reference output.

From the obtained execution trace, at each fault injection, we choose a random occurrence of a random instruction that updates at least one register. We use weighted random numbers to inject faults uniformly across the whole execution of a program. After the specific occurrence of an instruction is chosen, one of its output registers is randomly selected to inject a fault into. The injection of a fault is simulated by XORing the value of this register with a random integer. Such faults imitate both sporadic corruptions of CPU registers and miscomputations in CPU execution units. The fault occurs right after the selected instruction. Faults are injected in general-purpose registers, as well as in the status and x86-64-specific registers.

In the second step, we start the program under Intel SDE with GDB attached and inject a single fault. To inject a fault, we construct a GDB script to set a conditional breakpoint in the program based on the specified instruction address and its occurrence number. Whenever the breakpoint is triggered by any thread, the script injects a fault and resumes execution. After the program terminates, the output is examined to study the effect of the fault injection (see Table 1). The second step is repeated until a sufficient number of runs (fault injections) is reached.

Fault injection probabilistic model. Our fault injection tool injects only one fault per run and requires smallest inputs to finish one experiment in a reasonable amount of time. Hence, we also built a probabilistic fault injection framework to investigate reliability of HAFTed programs working for a longer time and under different fault rates. We use a probabilistic model checker tool called PRISM [41] to construct a continuous-time Markov chain model of HAFT (∼130 LOC) and verify its properties probabilistically. Figure 5 represents the model for the native, ILR, and HAFT architectures. The architectures differ in the transition rates, which are selected from our fault injection experiments (see §5.5).

The system starts with a correct state. A transient fault can transfer the system to a correct, corrupted, crashed, or HAFT-correctable state. If a system is not in a correct state, then it is unavailable and needs recovery. A crashed system can be recovered by rebooting, and a corrupted system by manual recovery. The system in a HAFT-correctable state is recovered by restarting a transaction; this state exists only in the HAFT architecture.

5. Evaluation

Our evaluation answers the following questions:

- What are the performance overheads of HAFT? (§5.2)
- How effective are the optimizations in improving the performance and reliability of HAFT? (§5.3)
- What is the effect of hyper-threading on HAFT? (§5.4)
- What is the level of fault tolerance achieved by HAFT, and how efficient is it under different fault rates on long-running programs? (§5.5)
- What is the code coverage provided by HAFT, i.e., what fraction of the run-time execution is protected? (§5.6)

5.1 Experimental Setup

Applications. We evaluated HAFT with applications from two multithreaded benchmark suites: Phoenix 2.0 [58] and PARSEC 3.0 [15]. We report results for all 7 applications in the Phoenix benchmark and 8 out of 13 applications in the PARSEC benchmarks. The remaining five applications are not supported for the following reasons: *bodytrack* and *raytrace* make use of C++ exceptions, which are currently not supported by our implementation; *freqmine* is an application based on OpenMP, which did not compile under our version of LLVM; *fluidanimate* produces nondeterministic output and thus makes it impossible to check the correctness of the results; and finally, the native version of *facesim* crashes with a runtime error when compiled with LLVM.

All applications were compiled with the HAFT compiler based on LLVM 3.7.0 with `-O3`, `-mrtm` (to support Intel TSX), and `-fno-builtin` (to transparently link against our own version of libc) flags and linked using the LLVM gold plugin.

Modified applications. Two applications from the Phoenix benchmark, *wordcount* and *kmeans*, have a high level of cache conflicts, which results in frequent transaction aborts. Therefore, we modified 47 LOCs in the former and 5 LOCs in the latter to mitigate this problem. We report results for both modified and unmodified versions. We refer to the modified ("no sharing") versions as *wordcount-ns* and *kmeans-ns*.

Datasets. For the performance evaluation, we used the largest available datasets provided by Phoenix and PARSEC benchmark suites. However, fault injection experiments were carried out using the smallest available input because they are extremely time consuming.

Testbed. We carried out the performance evaluation experiments on a machine with two 14-cores Intel Xeon processors operating at 2.0 GHz with hyper-threading enabled (Intel Haswell microarchitecture) with 128 GB of RAM, a 3.5 TB SATA-based SDD, and running Linux kernel 3.16.0. Each core has private 32 KB L1 and 256 KB L2 caches, and 14 cores share a 35 MB L3 cache. Due to hyper-threading, two logical threads sharing the same core also share the L1 and L2 caches. For fault injections, we used a cluster of 25 machines to parallelize the experiments.

Methodology. For all measurements, we confined our experiments to one processor, thus the maximum number of threads is restricted to 14 for all benchmarks. Note that we pinned application threads to separate physical cores in all experiments to avoid the effects of hyper-threading. In addition, we conducted an experiment to estimate how hyper-threading affects abort rates of HAFT (see §5.4).

For performance experiments, we ran programs with 1–14 threads. For fault injections, we fixed the number of threads to two. For each Phoenix benchmark, we performed a warm-up run to load input files into the main memory to stress-test the CPU overheads of HAFT (otherwise Phoenix benchmarks would be dominated by I/O). For PARSEC benchmarks, we reused the provided framework.

Measurements. For all performance measurements, we report the average over 10 runs. Fault injection experiments were conducted by injecting 2,500 faults for each program.

5.2 Performance Overheads

We first present the performance overheads of HAFT over the native execution. Figure 6 shows the overheads for a varying number of threads ranging from 1 to 14 threads.

The average overhead across all applications is 2× (see bar *mean*). The best case for HAFT is *matrixmul* due to the very low instruction-level parallelism (ILP) of 0.2 instructions/cycle for the native execution; thereby, HAFT effectively utilizes these spare ILP resources, with a runtime overhead of just 5%. The worst case for HAFT is *vips*, which incurs a slowdown of 4×, where two factors negatively affect HAFT's performance. First, the native version already has high ILP of 2.6 instructions/cycle such that there are no spare cycles left for HAFT. Second, *vips* has many calls to tiny functions such that the Tx local function calls optimization leads to a high performance penalty. If we disable this optimization, the performance overhead drops to 2.5× (*vips-nc* in Figure 6; see also next section).

HAFT benefits from the suboptimal scalability of native versions of programs. For example, the native version of *ferret* scales linearly, so the overhead of HAFT stays at the same level with the increasing number of threads. In contrast, the native version of *dedup* scales poorly with more than 2 threads and the overhead of HAFT is amortized in this case.

Table 2 (first three columns) highlights the contribution of HAFT components: ILR and Tx. ILR alone incurs performance

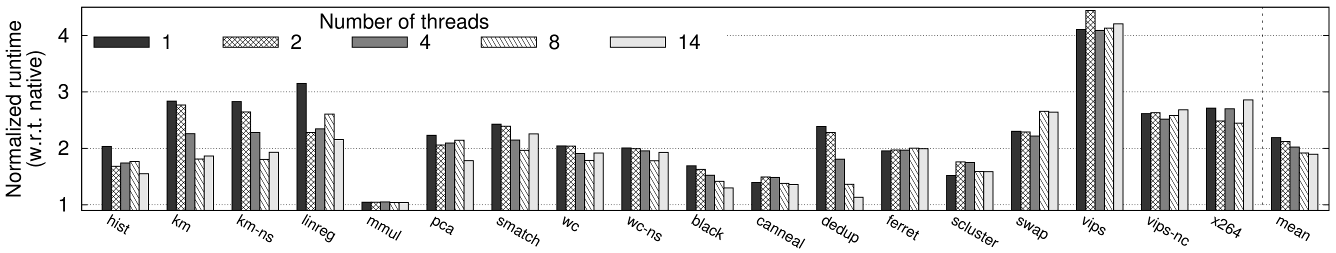

Figure 6: Performance overhead over native execution with the increasing number of threads (on a machine with 14 cores).

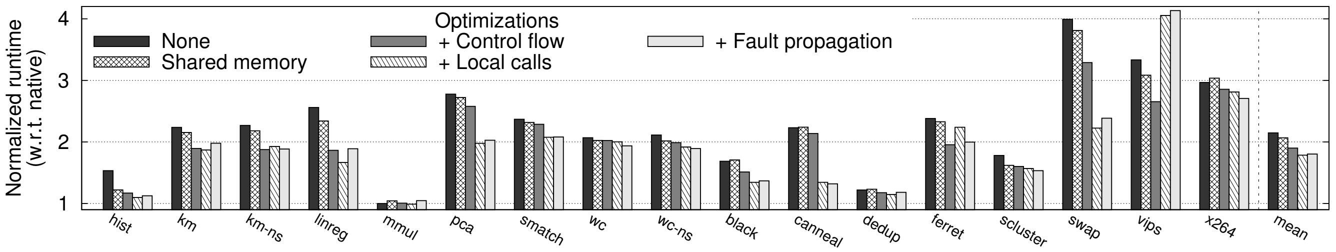

Figure 7: Performance overhead over native execution with different optimizations (with 14 threads).

Benchmark	Overheads (§5.2)			HT	Cov. %
	ILR	Tx	HAFT	(§5.4)	(§5.6)
histogram	1.46	1.02	1.55	1.0	95.7
kmeans	1.60	1.28	1.86	2.6	95.8
kmeans-ns	1.63	1.28	1.93	5.4	—
linearreg	2.03	1.12	2.16	1.2	97.2
matrixmul	1.04	1.01	1.04	377	88.9
pca	1.35	1.14	1.78	2.4	95.1
stringmatch	1.50	1.46	2.26	1.8	98.7
wordcount	1.35	1.39	1.92	1.5	95.1
wordcount-ns	1.45	1.31	1.93	8.9	—
blackscholes	1.17	1.06	1.30	2.9	93.9
canneal	1.16	1.13	1.36	1.3	67.6
dedup	0.99	1.02	1.13	1.1	75.1
ferret	1.32	1.25	1.99	12.6	96.9
streamcluster	1.46	1.18	1.59	1.9	92.7
swaptions	1.98	1.57	2.64	11.4	89.6
vips	2.16	2.29	4.21	1.5	85.1
vips-nc	2.19	1.46	2.68	1.3	—
x264	2.32	1.33	2.86	4.9	85.5
mean	**1.52**	**1.27**	**1.89**	**24.5**	**90.2**

Table 2: *First three columns*: Normalized runtime w.r.t. native of HAFT and its components (§5.2). *Fourth column*: Increase in abort rate when moving from the non-hyper-threaded to the hyper-threaded configuration (§5.4). *Fifth column*: Code coverage of HAFT in % (§5.6). All experiments with 14 threads.

overhead of 52% on average; this low overhead indicates that ILR efficiently uses spare ILP to hide additional instructions and checks inserted at compile-time. Tx incurs 27% overhead on average. Interestingly, the overhead of Tx is higher than that of ILR in the case of *vips*; as explained in the previous paragraph, this is due to the high number of calls to tiny functions. As soon as we remove this bottleneck, the overhead of Tx decreases by 60% (*vips-nc*).

5.3 Effectiveness of Optimizations

Impact of optimizations. The impact of different optimizations (§3.3) on performance is shown in Figure 7. We compare HAFTed benchmarks without any optimizations and then apply the following optimizations successively: ILR shared memory accesses, ILR control flow protection, Tx local function calls, and fault propagation check. Note that the fault propagation check is targeted to increase reliability at the price of some performance degradation.

This set of optimizations leads to an average performance improvement of 20% and in some cases achieves 70%. Interestingly, the addition of control flow checks, which are introduced to increase reliability, has a positive impact on performance: this happens because the check of a condition is substituted by a sequence of jumps, thus decreasing the number of executed instructions and benefiting from branch prediction.

Another somewhat surprising result is the Tx local function calls optimization: performance of most benchmarks improves significantly, whereas it degrades for *vips*. In the case of *vips*, the overhead of updating and checking the dynamic counter turns out to be higher than simply starting a new transaction on each function call. We decided to also show the results of *vips* with this optimization disabled (*vips-nc*) in other experiments.

Impact of transaction size. We show the impact of different transaction sizes (maximum number of instructions in one transaction) on the performance overhead and the number of aborts in Figure 8 respectively. Note that the number of threads is fixed to 14 in these experiments. Performance overhead decreases with greater transaction sizes, from 2.2× to 1.8× on average, due to the lower number of transactions. At the same time, the number of aborts grows with increasing transaction sizes. Aborts happen due to the following two reasons: first, longer transactions overflow the L1 cache more often, and second, longer transactions lead to higher probability of conflicts between threads.

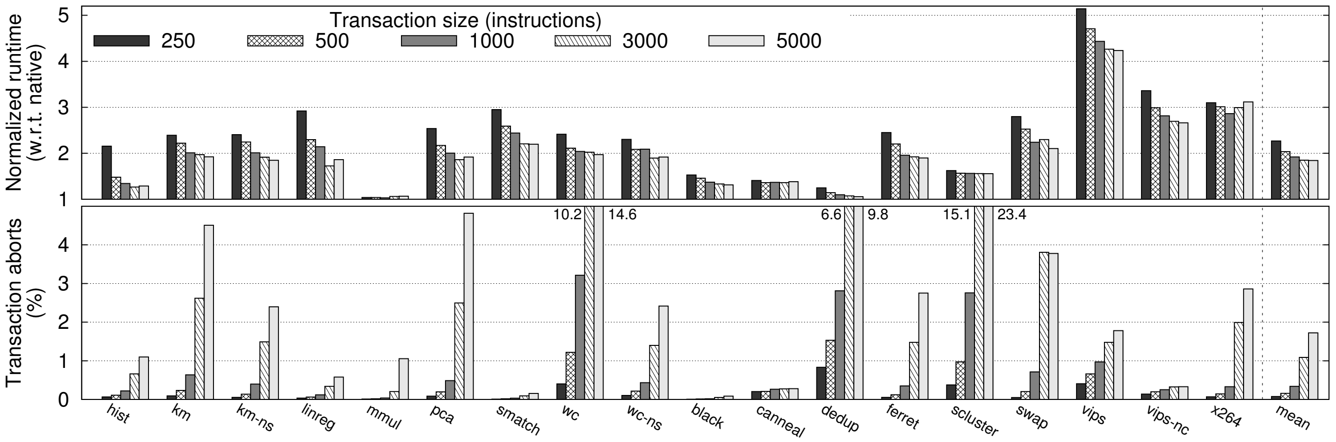

Figure 8: Performance overhead over native execution (top) and percentage of aborts (bottom) vs. transaction size (with 14 threads).

Benchmark	Abort rate (%)	Abort causes, % Capacity	Conflict	Other
histogram	1.10	0.48	30.16	69.36
kmeans	4.51	0.01	99.90	0.09
kmeans-ns	2.40	0.03	95.68	4.29
linearreg	0.58	0.00	0.13	99.87
matrixmul	1.05	66.21	0.06	33.73
pca	4.82	0.72	82.97	16.31
stringmatch	0.15	2.53	0.32	97.15
wordcount	14.60	1.27	94.90	3.83
wordcount-ns	2.42	16.24	20.80	62.96
blackscholes	0.08	2.20	0.50	97.30
canneal	0.28	1.34	2.70	95.96
dedup	9.84	16.29	1.50	82.21
ferret	2.75	80.40	0.62	18.98
streamcluster	23.40	0.11	99.89	0.00
swaptions	3.78	90.87	0.01	9.12
vips	1.78	40.40	41.75	17.85
vips-nc	0.33	2.36	97.64	0.00
x264	2.86	64.22	6.72	29.06

Table 3: Transaction abort rate and causes (with 14 threads). The worst-case transaction size of 5,000 is fixed for each benchmark.

Peculiarly, increasing transaction sizes (and thus higher abort rates) does not result in any clear pattern of performance overheads. Indeed, with increasing transaction sizes, two factors compete: (1) longer transactions amortize the cost of Tx instrumentation, and (2) the number of aborts increases because transactions start to overflow or conflict. The first factor decreases performance overhead while the second factor increases it.

This is evident from Figure 8. In the case of *streamcluster*, the number of aborts goes up to 23.4%, but longer (and fewer) transactions counterbalance this factor, and thus the performance overhead stays roughly the same. Compare it with *histogram*, where the number of aborts is low and the amortization factor dominates, thus decreasing the overhead. Finally, in the case of *x264*, the number of aborts drastically increases with transaction sizes greater than 1000, resulting in a change of the performance pattern.

The huge negative impact of cache sharing is clearly seen when comparing *kmeans* and *kmeans-ns* (removed true sharing), as well as *wordcount* and *wordcount-ns* (removed false sharing). In a demonstrative case of *wordcount*, rewriting the application with no cache sharing results in a 7× decrease of transaction aborts.

Table 3 shows the breakdowns of abort rates and their causes for each benchmark, measured with the worst-case transaction size of 5,000. The low abort rates (less than 1%) are largely dominated by the residual spontaneous ("other") aborts. Higher abort rates are caused either mostly by capacity overflows or by conflicts among simultaneous transactions. For example, all aborts in *kmeans* are due to high conflict rates, whereas *matrixmul* experiences many capacity overflows due to its cache-unfriendly behavior.

For all other plots, we set for each benchmark the transaction size to the greatest value such that the percentage of aborts is sufficiently low, in order to achieve the best trade-off between performance and reliability. For example, we set transaction size to 1000 for *kmeans* and *pca*, and to 5000 for *stringmatch* and *blackscholes*.

5.4 Effect of Hyper-threading

To estimate the effect of hyper-threading on HAFT, we conduct the experiment with 14 threads (similar to Figure 6, last bar). However, in this experiment we pin 14 logical threads to 7 physical cores. Thus, each pair of threads shares CPU execution units and L1 and L2 caches.

Table 2 (fourth column) highlights the increase in abort rates compared to the baseline configuration of 14 logical threads on 14 physical cores. Many benchmarks still have low abort rates (*histogram*, *linearreg*, *canneal*, etc.), but some exhibit dramatic increase in transaction aborts (*matrixmul*, *ferret*, *swaptions*, etc.). In the former case, transactions are sufficiently small to peacefully co-exist in the shared L1 cache. In the latter case, transactions compete for the limited capacity of the cache and abort each other.

The case of *matrixmul* is peculiar, with an abort rate increasing by 377× from negligible 0.07% aborts in non-hyper-threaded

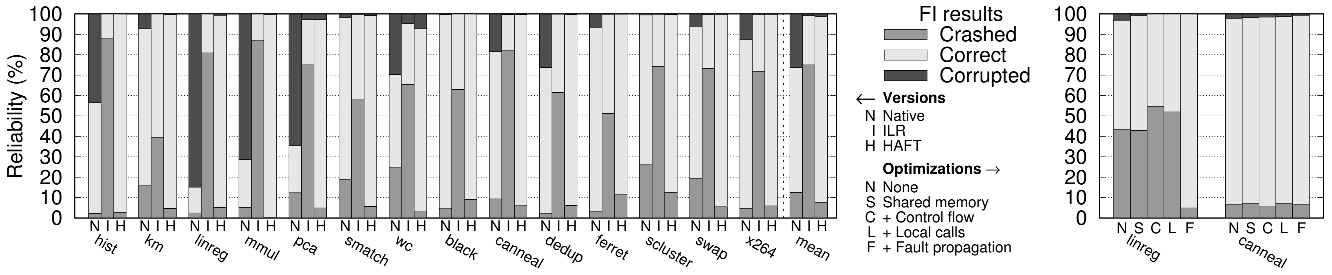

Figure 9: Reliability of HAFT (left) and impact of different optimizations on two benchmarks (right) with 2 threads.

Fault probabilities	Native	ILR	HAFT
Masked (%)	61.3	24.2	24.2
SDC (%)	26.2	0.8	1.1
Crashed (%)	12.5	75.0	7.7
HAFT-correctable (%)	—	—	67.0

Table 4: Parameters for the HAFT model.

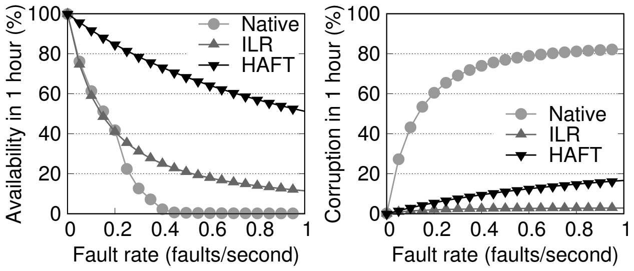

Figure 10: HAFT fault injection modeling. Plots show fractions of time when system is available (left) or corrupted (right) in a time span of one hour w.r.t. the fault rate.

scenario to 24% with hyper-threading. Our analysis indicates that aborts happen due to frequent overflows of a cache on read accesses – *matrixmul* is cache-unfriendly, and the sharing of L1- and L2-caches by two threads only exacerbates this problem.

5.5 Fault Injections

Fault injection experiments. The results of our fault injection experiments are shown in Figure 9. The faults were injected uniformly at random across the whole execution trace of each benchmark, including the parts not protected by HAFT (§4.1). Note that we were not able to perform fault injections into *vips* as injecting one fault under Intel SDE took more than an hour even under the smallest inputs.

We also performed the experiment where the faults were injected only in the protected parts of the benchmarks, with very similar outcomes. This is expected: Our statistics indicates that the faults landing in unprotected parts constitute less than 1% of all injected faults in almost all cases except for *wordcount* and *x264*. Therefore, we do not show the results of this experiment.

Even in native execution, most of the faults (61.3%) are masked and programs remain correct after a fault is injected. However, around 26% of faults lead to data corruptions (see bar *mean*). When applying ILR, almost all faults (99.2%) are detected, but programs exit prematurely 75% of the time, leading to low availability (this can be explained by the fact that ILR sometimes detects also those faults that would be masked in native execution). Finally with HAFT, program reliability increases to approximately 91.2%. (Program reliability with HAFT reaches 92% on average if the faults are injected only in the protected parts of benchmarks.)

Figure 9 (right) shows the impact of different optimizations on the reliability of HAFT. As conducting these experiments is highly time-consuming, we chose only one benchmark from Phoenix (*linearreg*) and one from PARSEC (*canneal*). Note that the non-optimized versions (N) have a non-negligible number of data corruptions. In the case of *canneal* optimizations only slightly decrease the number of data corruptions, while for *linearreg* the shared memory optimization (S) and the addition of control flow protection (C) lead to SDC-free executions, but also slightly increase the proportion of crashes. The local calls optimization (L), which is only intended for performance improvement, has no effect on reliability. Finally, the fault propagation check (F) improves the availability of *linearreg* dramatically, reducing the number of crashes from 50% to less than 5%.

Fault injection modeling. To measure the reliability of HAFT, we use the model from §4.2 and parameters from Table 4. Fault probabilities are extracted from the fault injection experiments. We choose the following recovery rates: 6 hours for manual recovery, 10 seconds for machine reboot, and 2.5 μs for transactional recovery in HAFT. The rate of manual recovery is based on the Amazon report where it took 6 hours between the first noticed corruption and the renewal of processing of requests [1]. The rate of machine reboot is based on the time needed for a complete reboot of our server. The rate of HAFT recovery is based on the maximum transaction size of 5,000 instructions, which corresponds to the maximum latency of recovery of 2.5 μs on a 2.0 GHz CPU.

Figure 10 (left) shows the fraction of time when the system is available in a time span of one hour with regard to the fault rate. The fault rate varies from once every hour to once every second (0.00028 to 1 fault/second). HAFT significantly increases program availability compared to ILR and native. For example, under a fault rate of 1.0, HAFT's availability is around 50%, i.e., 30 minutes out of one hour. In contrast, availability of native and ILR versions is 0% and 10% (6 minutes) respectively. In addition, Figure 10 (right) indicates that ILR and HAFT drastically reduce the number of data corruptions. Native spends more than 80% of the time in a corrupted state, while both ILR and HAFT stay in this state for less than 20%.

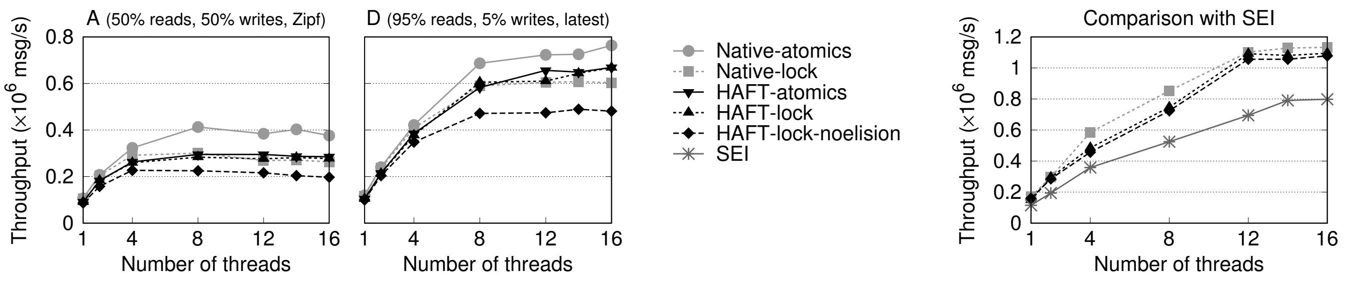

Figure 11: Memcached throughput. Left two graphs: workloads A and D. Right graph: comparison of HAFT and SEI using a mcblaster client, a key range of 1,000, and values of size 128 B (same experimental setup as in [11]).

5.6 Code Coverage

Lastly, we analyzed what fraction of the run-time execution is protected with HAFT. Remember that our prototype of HAFT does not protect external libraries except for partial support of libc (see §4.1). To this end, we measured the fraction of dynamic execution spent inside transactional execution (Table 2, fifth column). The fraction is calculated as the number of cycles executed in transactions to the number of all cycles executed, as reported by the perf tool. Each program was built with all HAFT optimizations enabled and with the number of threads fixed to 14; the number of retries was set to three. The mean code coverage across all benchmarks is 90.2% indicating a high level of protection for almost all applications. Two exceptions are *canneal* and *dedup*: the former extensively uses containers from libstd++ while the latter spends many cycles in unprotected parts of libc for thread management and dynamic allocation.

6. Case Studies

We successfully applied HAFT on five real-world applications without any source code modifications. Due to space limitations, we present detailed results only for Memcached (§6.1) and present summarized results for the others (§6.2). All applications were run in a local deployment on a single Haswell machine: we deployed each server application on one 14-core processor and its client applications on the other processor.

6.1 Memcached Key-Value Store

We evaluated Memcached [27] v1.4.24 using workloads from the YCSB benchmark [21] with 1 million key-value queries, each key being 16 B and each value 32 B. Figure 11 (left two graphs) shows the throughput of Memcached increasing with the number of threads, with two extreme YCSB workloads corresponding to the best and worst case for HAFT: A (50% reads, 50% writes, Zipf distribution) and D (95% reads, 5% writes, latest distribution). We evaluated Memcached with all available variants for synchronization using pthreads locks and atomic operations. For both native and HAFT, we tested two versions, one with locks only (native-lock and HAFT-lock) and one with atomics enabled (native-atomics and HAFT-atomics). Note that HAFT-lock has the optimization of lock elision (see §3.3). We also show the version with this optimization disabled (HAFT-lock-noelision).

The lock elision optimization allows HAFT-lock to perform 30% better than HAFT-lock-noelision and on par with native-lock, i.e., the overhead of HAFT is completely amortized by this optimization. Indeed, when configured to use locks, Memcached spends most of the time acquiring and releasing the locks. Since HAFT already uses transactions for recovery, removing the overhead of these locks comes for free. Moreover, HAFT-lock performs similar to HAFT-atomics, indicating that an application can achieve the same performance improvement with lock elision as when using atomics.

Our experiments also show that the latency of HAFT is 30% worse than in native on average and the percentage of committed transactions remains above 95% in all runs. Finally, the fault injection experiments indicate that HAFT decreases the percentage of data corruptions from 2% to 0.09% (two SDCs). Both lingering data corruptions happened in the very beginning of two functions responsible for shaping a reply message (namely, `add_bin_header` and `add_iov`). In both cases, the "length" function argument was corrupted exactly before its shadow copy was created; as a result, the reply string was incorrectly truncated.

Comparison with SEI. We also compared HAFT against SEI [11], another state-of-the-art approach, using Memcached.[4] We deployed SEI locally on our Haswell machine and reproduced the experiments from the SEI paper with the mcblaster client, a key range of 1,000, and values of size 128 B. Since SEI performs modifications to Memcached, we apply HAFT on the modified version.

Figure 11 (right graph) shows that HAFT performs on par with the native version (similar to graphs on the left) and outperforms SEI by 30–40%. The lower performance of SEI is explained by the local deployment; in the experiments with remote clients in the original paper [11], SEI's overhead was amortized by the network. Also note that the lock elision optimization of HAFT provides no benefit in this experiment. This is due to an older version of Memcached (namely, version 1.4.15) that supports only coarse-grained locks and thus is not amenable to our simple lock-elision heuristics.

We conclude with an indirect comparison of fault coverage, based on the numbers reported in [11].[5] As shown earlier, HAFT leaves 0.09% of data corruptions, whereas SEI with a similar configuration cannot detect 0.15% of corruptions.

[4] Note that Memcached is the only multithreaded application evaluated in [11].

[5] These numbers should be treated with care because of the differences in Memcached versions, fault models and fault injection frameworks used.

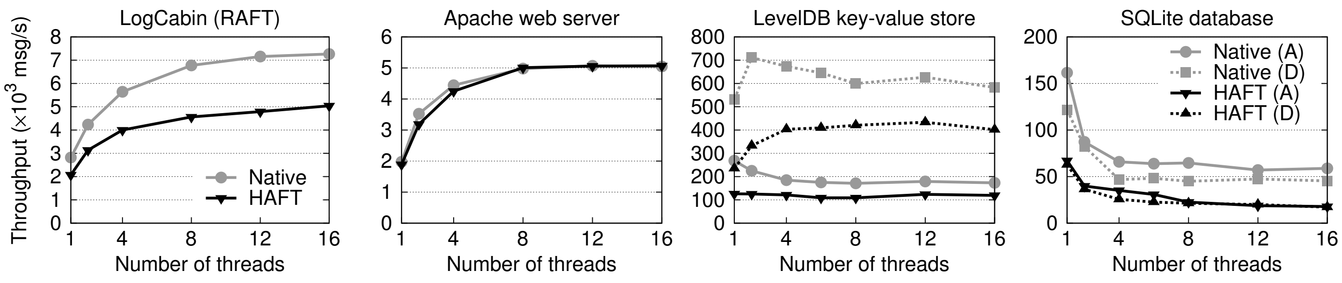

Figure 12: Throughput of additional case-studies: LogCabin (RAFT), Apache web server, LevelDB key-value store, and SQLite database. Two extreme workloads are shown for LevelDB and SQLite: workload A (50% reads, 50% writes, Zipf distribution) and workload D (95% reads, 5% writes, latest distribution).

6.2 Additional Case-Studies

LogCabin (RAFT). LogCabin [7] is an implementation of a consistent storage mechanism built on the RAFT [52] consensus protocol. For the evaluation, we used the benchmark shipped together with LogCabin that repeatedly writes 1,000 values into a memory-mapped file.

Apache web server. Apache is a popular web server [2]. For multithreading, we use a "worker multi-processing module" with a single running process and a varying number of worker threads. We used the Apache *ab* benchmark tool that queries a static 1 MB Web page for the evaluation.

LevelDB key-value store. LevelDB is a fast embedded key-value storage library developed by Google [5]. We evaluated LevelDB on an in-memory database using the same YCSB workloads used for Memcached (workloads A & D).

SQLite database. SQLite is an SQL database engine implemented as an embeddable software library [53]. We evaluated SQLite on an in-memory database using again YCSB workloads A and D.

The scalability plots are shown in Figure 12. LogCabin and LevelDB are well-behaved applications, performing 25–35% worse than native versions. Apache exhibits an overhead of just 10%; this good result is due to Apache's extensive use of external libraries that are not transformed by HAFT. SQLite shows the poorest results, with HAFT performing 3–4× worse than the native version. We attribute this poor performance mainly to the extensive use of function pointers that are conservatively treated as external functions by HAFT.

We performed fault injection experiments on LevelDB and SQLite. Though their native versions are already tolerant to data corruptions, the faults lead to a high number of crashes, 42% and 28% respectively. HAFT decreases these numbers to only 10% and 3.7%, providing significantly higher availability.

7. Conclusion and Future Work

Many software systems require very high level of reliability. Alas, adding fault tolerance capabilities to existing applications inevitably degrades their performance. Fortunately, modern commodity hardware with its increased instruction level parallelism and new extensions such as hardware transactional memory enables cheap and efficient fault tolerance solutions. In this paper, we presented HAFT, a novel approach to software hardening that provides low-cost fault detection via instruction-level redundancy and fast fault recovery via HTM. Our evaluation shows that HAFT significantly increases reliability and availability at the cost of 2× performance overhead.

Future Work. In the current design of the transactification algorithm, a single threshold value is chosen for the entire execution of a program (§3.2). In reality, different code paths of the same program exhibit different behavior with respect to hardware transactions. In this case, some form of static/dynamic adjustment of the threshold could prove beneficial.

Our current implementation of HAFT does not protect all program code. While adding protection to the most of the functionality that standard libraries provide seems straightforward, supporting inline assembly and the C++ exception mechanism would require substantial engineering effort. Another problem is unfriendly instructions which inevitably lead to TSX transaction aborts. We believe this can be fixed in the future implementations of TSX. Fortunately, once these issues are resolved, all programs written in LLVM-backed programming languages could be transparently hardened.

Hardware transactional memory can be found in architectures other than x86-64. For example, IBM POWER8 [18] provides not only regular TSX-like transactions, but also *rollback-only transactions* which buffer stores without detecting data conflicts. Moreover, transactions in POWER8 can be suspended and resumed to avoid aborting on interrupts. We are currently investigating how these properties can benefit HAFT.

Software availability. HAFT's source code is publicly available under `https://github.com/tudinfse/haft`.

Acknowledgements. We thank Norman Rink for his help with the LLVM compiler backend. We also thank Manos Kapritsos, Sergei Arnautov, Maksym Planeta, our anonymous reviewers, and our shepherd Andreas Haeberlen for their helpful comments. This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed).

References

[1] Amazon S3 availability event. http://status.aws.amazon.com/s3-20080720.html. Accessed: Oct, 2015.

[2] Apache HTTP server project. http://httpd.apache.org/. Accessed: Oct, 2015.

[3] Data corruption with Opteron CPUs and NVidia chipsets. https://bugzilla.kernel.org/show_bug.cgi?id=7768. Accessed: Oct, 2015.

[4] Intel Software Development Emulator (Intel SDE). https://software.intel.com/en-us/articles/intel-software-development-emulator. Accessed: Oct, 2015.

[5] LevelDB key-value storage library. https://github.com/google/leveldb. Accessed: Oct, 2015.

[6] LLVM atomic instructions and concurrency guide. http://llvm.org/docs/Atomics.html. Accessed: Oct, 2015.

[7] LogCabin distributed storage system. https://github.com/logcabin/logcabin. Accessed: Oct, 2015.

[8] musl libc. http://www.musl-libc.org/. Accessed: Oct, 2015.

[9] New defective S3 load balancer corrupts relayed messages. https://forums.aws.amazon.com/thread.jspa?threadID=22709. Accessed: Oct, 2015.

[10] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data System Research (CIDR)*, 2011.

[11] D. Behrens, M. Serafini, S. Arnautov, F. P. Junqueira, and C. Fetzer. Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[12] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. In *proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[13] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed. Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[15] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.

[16] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. In *IEEE Micro*, 2005.

[17] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[18] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for Transactional Memory in the Power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[19] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[20] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[22] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[23] H. Cui, R. Gu, C. Liu, T. Chenx, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[24] B. Döbel and H. Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, 2014.

[25] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[26] C. Fetzer and P. Felber. Transactional memory for dependable embedded systems. In *Proceedings of the 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011.

[27] B. Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124), Aug. 2004.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, 1990.

[29] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[30] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, and D. Zhou. Rex: Replication at the Speed of Multi-core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014.

[31] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[32] F. Haas, S. Weis, S. Metzlaff, and T. Ungerer. Exploiting Intel TSX for fault-tolerant execution in safety-critical systems. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014.

[33] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[35] M. Isard. Autopilot: Automatic data center management. *SIGOPS Operating Systems Review*, 41(2):60–67, Apr. 2007.

[36] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[37] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[38] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. Improving server applications with system transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[39] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[40] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer. Elzar: Triple Modular Redundancy using Intel AVX. In *proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2016.

[41] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic model checking for performance and reliability analysis. *SIGMETRICS Performance Evaluation Review*, 36(4):40–45, Mar. 2009.

[42] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, C-28(9):690–691, Sept 1997.

[43] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[44] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)*, 2014.

[45] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the 23d ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[46] P. E. McKenney, M. M. Michael, J. Triplett, and J. Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. *SIGOPS Operating Systems Review*, 44(3):93–101, Aug. 2010.

[47] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[48] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011.

[49] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[50] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[51] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[52] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[53] M. Owens and G. Allen. *SQLite*. Springer, 2010.

[54] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015.

[55] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[56] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[57] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (Micro)*, 2001.

[58] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[59] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. In *IEEE Micro*, 2007.

[60] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *proceedings of the 3th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2005.

[61] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer. An experimental study of soft errors in microprocessors. In *IEEE Micro*, 2005.

[62] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. FailStar: Towards a versatile fault-injection experiment framework. In *Proceedings of the Architecture of Computing Systems (ARCS)*, 2012.

[63] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[64] B. Schroeder, G. Gibson, et al. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[65] M. Shafique, S. Garg, J. Henkel, and D. Marculescu. The EDA challenges in the Dark Silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, 2014.

[66] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2002.

[67] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[68] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2006.

[69] Y. J. Song, F. P. Junqueira, and B. Reed. BFT for the skeptics. In *BFTW3, affiliated with DISC*, 2009.

[70] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[71] G. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Verissimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan 2013.

[72] C. Wang, H. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *proceedings of the 5th International Symposium on Code Generation and Optimization (CGO)*, 2007.

[73] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014.

[74] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[75] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[76] G. Yalcin, A. Cristal, O. Unsal, A. Sobe, D. Harmanci, P. Felber, A. Voronin, J.-T. Wamhoff, and C. Fetzer. Combining error detection and transactional memory for energy-efficient computing below safe operation margins. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2014.

[77] G. Yalcin, O. S. Unsal, and A. Cristal. Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2013.

[78] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel Transactional Synchronization Extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[79] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August. Runtime asynchronous fault tolerance via speculation. In *proceedings of the 10th International Symposium on Code Generation and Optimization (CGO)*, 2012.

[80] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled acyclic fault tolerance. In *proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.