ELZAR: Triple Modular Redundancy using Intel AVX

(Practical Experience Report)

Dmitrii Kuvaiskii† Oleskii Oleksenko‡ Pramod Bhatotia† Pascal Felber‡ Christof Fetzer‡
† Technical University of Dresden, Germany
‡ University of Neuchâtel, Switzerland

Abstract—Instruction-Level Redundancy (ILR) is a well-known approach to tolerate transient CPU faults. It replicates instructions in a program and inserts periodic checks to detect and correct CPU faults using majority voting, which essentially requires three copies of each instruction and leads to high performance overheads. As SIMD technology can operate simultaneously on several copies of the data, it appears to be a good candidate for decreasing these overheads. To verify this hypothesis, we propose ELZAR, a compiler framework that transforms unmodified multithreaded applications to support triple modular redundancy using Intel AVX extensions for vectorization. Our experience with several benchmark suites and real-world case-studies yields mixed results: while SIMD may be beneficial for some workloads, e.g., CPU-intensive ones with many floating-point operations, it exposes higher overhead than ILR in many applications we tested.

I. INTRODUCTION

Transient faults in CPUs can cause arbitrary state corruption during computation. Therefore, they pose a significant challenge for software systems reliability [1]. The causes for transient faults are manifold, including radiation/particle strikes, dynamic voltage scaling, manufacturing variability, device aging, etc. [2]. Moreover, the general trend of ever-decreasing transistor sizes with lower operating voltages only worsens the reliability problem [3].

The unreliability of CPUs is especially threatening at the scale of data centers, where tens of thousands of machines are used to support modern online services. At this sheer scale, CPU faults happen at a surprisingly high rate and tend to increase in frequency after the first occurrence, as reported by a number of large-scale in-the-field studies [4], [5]. Since the machines in data centers operate in tight collaboration, a single CPU fault can propagate to the entire data center, leading to catastrophic consequences [6].

To overcome the problem of transient CPU faults, a number of light-weight hardening techniques were proposed. These hardening techniques transform the original program to locally detect and correct faults. A well-known hardening approach is Instruction-Level Redundancy (ILR) [7], [8]. ILR is a compile-time transformation that replicates original instructions to create separate data flows and inserts periodic checks to detect divergence caused by transient faults in these data flows. In particular, ILR duplicates instructions to achieve fault detection [7], [8] and triplicates them to tolerate faults by majority voting [9].

As a result, with ILR the CPU executes the same instruction two or three times on several data copies. We notice that, in fact, this corresponds to the very definition of Single Instruction Multiple Data (SIMD) processing. SIMD exploits data level parallelism, i.e., a single instruction operates on several pieces of data in parallel. Given that most modern CPUs have support for SIMD processing (Intel x86’s SSE and AVX, IBM Power’s Altivec, and ARM’s Neon), we can naturally ask the following question: Can we utilize SIMD instructions to tolerate transient CPU faults and achieve better performance than ILR with three copies?

To this end, we propose ELZAR, a compiler framework to harden unmodified multithreaded programs by leveraging SIMD instructions available in modern CPUs (§II). ELZAR is built on the Intel AVX technology to achieve triple modular redundancy, a classical approach for achieving fault tolerance in mission-critical systems. TMR detects faults by simple comparison of three replicas and performs fault recovery by majority voting, i.e., by detecting which replica differs from the other two and correcting its state. Consequently, it imposes an obvious restriction on the fault model: only one replica is assumed to be affected by the fault.

Since AVX possesses 256-bit wide registers and regular programs operate on at most 64-bit ones, it is possible to operate with four replicas in parallel, which is more than enough to harden applications and mask faults with majority voting. Consequently, if a hardware fault affects one of the four replicas in an AVX register, it can be detected and outvoted by the other, correct replicas.

We implemented ELZAR as an extension of the LLVM compiler framework. In particular, ELZAR transforms all the regular instructions of an application into their AVX-based counterparts, replicating data throughout AVX registers.

We evaluated our approach by applying ELZAR to the Phoenix and PARSEC benchmark suites (§III), as well as three real-world case-studies: Memcached, SQLite3, and Apache (§IV). To our disappointment, our evaluation showed mostly negative results, with an average normalized runtime slowdown of 4.1–5.6× depending on the number of threads. When compared against a straightforward instruction triplication approach [9], ELZAR performed 46% worse on average. At the same time, ELZAR was better on CPU-intensive benchmarks with few memory accesses and many floating-point operations.

We attribute poor performance of ELZAR to two main causes. First, there is a significant discrepancy between the regular CPU instructions and their AVX counterparts. This discrepancy forced us to introduce additional wrapper instructions that significantly hamper performance. Second, AVX instructions in general have higher latencies and are less optimized than the regular CPU instructions.

1Named after a four-armed character of Futurama. Similarly, Intel AVX has 4×64-bit wide registers for SIMD processing.
II. DESIGN AND IMPLEMENTATION

In this section, we first briefly introduce the AVX technology. We then present the design of ELZAR and describe the principle of ILR upon which it is based. We finally discuss its implementation and the fault injection framework.

A. Intel AVX

Our solution relies heavily on the Single Instruction Multiple Data (SIMD) technology and its specific implementation, Intel AVX. The main idea behind it is to perform the same operation on multiple pieces of data simultaneously (data level parallelism). Figure 1 illustrates this concept and how it relates to replication for fault tolerance. AVX adds new wider registers (YMM registers) that are capable of storing several elements and the corresponding new instructions that operate on these elements in parallel. Initially, AVX was targeted for applications that perform parallel data processing such as image or video processing; in this work, we (ab)use it for fault recovery. Note that we do not use the previous generation of Intel’s SIMD implementation, SSE, since it can only operate on two 64-bit values and we need at least three copies to be able to correct faults.

![Fig. 1: Addition in AVX. The original values r1 and r2 are replicated throughout the AVX registers. All four copies are computed in parallel.](image1)

The x86-64 architecture provides 16 256-bit wide YMM registers available for AVX instructions. It should be noted, however, that even though only 16 registers are visible at the assembly level, many more registers are implemented physically and used at runtime (e.g., 168 YMM registers in Intel Haswell).

The AVX instruction set consists of a large number of instructions, including special-purpose extensions for cryptography, multimedia, etc.

Most arithmetic and logic operations are covered by AVX, except for integer division and modulo. For example, Figure 1 illustrates how addition is performed with AVX. AVX-based comparisons act differently than their counterparts in the general instruction set. Instead of directly affecting the flags in the x86 FLAGS register as normal comparisons do, AVX comparisons return either all-1 (if result is “true”) or all-0 (“false”) values for each YMM element. This behavior is explained by the fact that the comparison is performed in parallel on multiple pieces of data, with possibly conflicting outcomes that would affect the flags differently. On the other hand, there are no control flow instructions in the general instruction set that could operate on such sequences of 1s and 0s. Therefore, a ptest AVX instruction was introduced that sets the ZF and CF flags in FLAGS by performing an and/andn operation between its operands. As a result, a branch is encoded in AVX as a sequence of an AVX comparison followed by a ptest and a subsequent jump based on the ZF and CF flags.

In this work, we use shuffle, a specific AVX operation that performs data rearrangement inside a YMM register. One example of a shuffle is shown in Figure 2. In combination with other operations, it allows us to get much of the functionality that is not implemented in hardware. For example, we can get a horizontal test for equality using a combination of shuffle, xor and ptest (see §II-D for more details).

B. System Model

Fault model. ELZAR uses the Single Event Upset (SEU) fault model [8], where only one bit-flip in a CPU is expected to occur during the whole execution of a program. A bit-flip means an unexpected change in the state of a CPU register or a wrong result of a CPU operation. The SEU is transient, i.e., it does not permanently damage the hardware and lasts only for several clock cycles.

We fully protect the AVX register file and the AVX operations; recall that they are completely decoupled from the regular GPR registers and scalar instructions (§II-A). We do not consider faults in the memory subsystem since it is assumed to be protected by ECC. Our fault model also does not cover control flow errors, assuming some orthogonal control flow checker.

In general, ELZAR protects from more than single faults. Indeed, four copies of data can tolerate two independent SEUs with a high probability: If any two copies agree and each of the other two copies disagree with the former ones, the majority voting can still mask the faults in the latter copies (we elaborate more on that in §II-D). In what follows, we focus on tolerating single faults for simplicity.

Memory and synchronization model. ELZAR imposes no restriction on the underlying memory and synchronization model, and even works with programs containing data races. ELZAR does not replicate nor modify the original memory-related operations (loads, stores, atomics) in any way, therefore the program’s memory access behavior is unchanged. As a result, ELZAR allows for arbitrary thread interleavings in multithreaded programs and supports all kinds of synchronization primitives.

C. Instruction-Level Redundancy

We base ELZAR on Instruction-Level Redundancy (ILR) [7], [8], [9], a software-based technique to detect and tolerate transient hardware faults. As other software-based approaches, ILR transforms the original program by replicating its computation and inserting periodic checks on computation results. An example of an ILR-transformed code snippet is shown in Figure 3b.

Replication. ILR replicates programs at the level of instructions. At compile-time, ILR inserts “shadow” copies for each instruction except for a few instructions classified as “synchronization” instructions. The shadow copies operate on their own set of shadow registers. At runtime, the program effectively executes the original and the shadow instructions, creating mostly independent original and shadow data flows which synchronize only on specific instructions.

The synchronization instructions include all memory-related operations (loads, stores, atomics) and control-flow operations (branches, function calls, function returns). Memory-related

---

2We omit the detailed explanation of how ptest works for the sake of simplicity. We refer the reader to the Intel architecture manuals.
operations are not replicated for two reasons: (a) the memory
subsystem contains only one copy of the state and there is
no need to store twice, and (b) ILR keeps the memory access
behavior unmodified in order to allow for non-determinism
in multithreaded applications. Control-flow operations are
not replicated because ILR protects only data integrity and assumes
no control-flow faults. Note that by not replicating function
calls, ILR requires no changes in function signatures and no
wrappers for system calls and third-party non-hardened libraries.

To create a shadow data flow, ILR replicates all inputs:
values loaded from memory, values returned by function calls,
and function arguments. This is achieved by a simple move
of an input value in one of the shadow registers.

If only fault detection is required, it is sufficient to duplicate
the instructions and signal an error or simply crash if two data
flows diverge [7], [8]. If fault tolerance is needed, the instructions
must be triplicated and majority voting must be used to
mask faults in one of the three data flows (see Figure 3b) [9].

Checks. To be able to detect faults, ILR additionally inserts
checks right before synchronization instructions. As one
example, a load address must be checked before the actual
load, otherwise a wrong value could be undetectably loaded
and used by the subsequent instructions. As another example,
all function arguments must be checked before the function
call to prevent the callee from computing with wrong values.
Finally, it is important to check the branch condition before
branching or else the program could take a wrong path.

The checks themselves are straightforward. If crash-stop
behavior is sufficient, a check compares two copies of data
and crashes the program if the copies diverge. For availability
(fault tolerance), ILR requires majority voting on three
replicas to mask a possible fault (as depicted in Figure 3b).
Moreover, the A VX-based
replicas to mask a possible fault (as depicted in Figure 3b).
Much replication.

Step 1: Replication. AVX provides an almost complete set
of arithmetic and logical instructions: addition, subtraction,
multiplication, bit-wise operations, shifts, etc. For floating
point data, all the usual instructions are present in AVX. For
integers, the only missing instructions are integer division and
modulo; ELZAR falls back to basic ILR in these cases. In
general, ELZAR achieves replication by simply replacing
the original arithmetic and logical instructions with their AVX
counterparts, as in Figure 1.

The situation is more complicated for (most) non-replicated
synchronization instructions. These are the regular loads,
stores, function calls, etc., which do not operate on YMM
registers. Thus, ELZAR has to extract one copy of each
instruction’s argument from YMM registers and use this copy
in the instruction. If a synchronization instruction returns a
value (e.g., load), this value must then be replicated inside a
YMM register. AVX provides dedicated instructions for such
purposes: extract and broadcast. Unfortunately, these
additional instructions must wrap every single load, store, etc.,
which leads to high overheads. An example of such “wrapping”
for a load is shown in Figure 4.

Fig. 4: Loads in ELZAR. The original load is wrapped by
AVX-based extract and broadcast.

A special case of a synchronization instruction is a branch.
A typical x86 branching sequence consists of one comparison
(cmp) which toggles the FLAGS register and the subsequent
jump instruction (je for “jump if equal”, jne for “jump if not
equal”, etc.). This is exemplified in Lines 7–10 of Figure 3a.
Unfortunately, as explained in §II-A, AVX lacks instructions
affecting control flow except for ptest. Moreover, the AVX-
based comparison instructions (e.g., cmpeq) do not toggle the
FLAGS register but instead fill the elements of a YMM register
with true/false values. Therefore, ELZAR inserts an additional
ptest to examine the result of cmpeq and only then proceeds
to a jump (see Figure 5 and also Figure 3c, Lines 7, 8, and 10).

ELZAR, on the other hand, does not replicate instructions
but rather data and thus increases the total number of
instructions only modestly. Figure 3c shows that ELZAR inserts
only 2 additional instructions to perform a check on a branch
condition. The replication is achieved by utilizing wide YMM
registers, with y1−y4 each containing four copies of the
original values. The add and cmp instructions in this snippet
are actually AVX instructions which operate on four copies
inside the YMM registers in parallel. The somewhat peculiar
check consists of the ptest AVX instruction and a subsequent
jump to recovery code if a discrepancy in branch condition
y4 is detected; we cover AVX-based checks in detail below.

In general, ELZAR transforms a program as follows: it
(1) replicates the data in YMM registers, (2) inserts periodic
checks, and (3) inserts recovery routines. In the following, we
discuss each of these steps in detail.

D. ELZAR

As appears clearly in Figure 3, ILR requires three times
more instructions than the original program plus expensive
majority voting on synchronization events. As a result, a
simple 3-instruction loop may require around 13 instructions
under ILR. Such a blow-up in instructions can quickly saturate
CPU resources and result in high performance overhead.

Fig. 3: Original loop (a) increments r1 by r2 until it is equal
to r3. Usual ILR transformation (b) triplicates instructions
and adds majority voting before comparison. AVX-based
ELZAR (c) replicates data inside YMM registers, inserts
ptest for comparison, and jumps to majority voting only
if a discrepancy is detected in y4.
Fig. 5: Branching in ELZAR. The original cmp for equality is transformed in a sequence of cmpeq and ptest.

**Step 2: Adding checks.** In order to detect faults, ELZAR inserts checks before each synchronization instruction. If a check succeeds, i.e., all copies of a YMM register contain the same value, the program continues normally, otherwise the YMM register must be recovered via majority voting. Note that the check itself must be as efficient as possible since it executes on the fast path. The recovery routine, however, resides on the slow path and can hence be less efficient.

Similar to replication, ELZAR distinguishes between branches and all other synchronization instructions. Because of implementation choices in AVX, checks turn out to be very effective for branches but not for other operations. To support efficient checks in ELZAR, we rely on the assumption that a fault corrupts only one copy in a YMM register (see §II-B).

In general, a check on the arguments of a synchronization instruction requires a pair-wise (horizontal) comparison of copies inside a YMM register. For example, upon a function call, all function arguments replicated in the corresponding AVX registers must be checked for discrepancies. Interestingly, AVX provides a horizontal subtraction instruction called hsub, but it is not implemented for 64-bit integers and is generally slow. Hence, we opted for another implementation of checks that involves a shuffle and a subsequent xor. This idea is illustrated in Figure 6. In an error-free case, xor produces all-0s which is easily ruled out by ptest. In the case of a fault in one of the copies, the result of xor is a mix of 0s and 1s, which triggers the jne path and leads to recovery.

**Step 3: Adding recovery.** Checks on branches and other synchronization instructions trigger a recovery routine when a fault is detected. The task of this routine is to mask a fault. Because of the assumption that a fault is localized in only one copy of the YMM register (see §II-B), it is sufficient to identify two identical replicas in the register and blindly broadcast their value to the whole register. This can be performed efficiently by a single comparison of the low elements of the faulty YMM register (depicted in gray in Figures 6 and 7) and, depending on the result of the comparison, copying either the lowest or the highest element to the rest of the register.

We note, however, that we could easily implement a smarter recovery strategy that would support more complex fault patterns involving multiple bit flips. As the recovery procedure is on the slow path, i.e., it is triggered only rarely, it does not need to be optimized for speed and this added reliability can be implemented without compromising performance.

E. Implementation

We developed ELZAR as a compiler pass in LLVM 3.7.0 [10] (~600 LOC). Additionally, we extract the implementation of checks and recovery in a separate LLVM IR file (~250 LOC). This separation allowed us to write the pass in a (mostly) target-independent way, i.e., AVX can be substituted by another similar technology (e.g., ARM Neon) by rewriting only the IR file with checks and recovery.

In order to be able to use AVX for replication, we disallow any vectorization in original programs. All other optimizations are enabled. Additionally, we run the scalarrepl pass to replace all aggregate data types ( structs, arrays) because they are not natively supported by LLVM vectors we employ.

**F. Fault Injection Framework**

For time budget reasons, we ran our fault injection experiments on a medium-sized cluster of computers without AVX installed. We therefore reused a fault injection tool from [11] which provides support for AVX instructions and gdb debugger. The tool injects one fault per program run and classifies the outcomes as listed in Table I.

<table>
<thead>
<tr>
<th>FI outcome</th>
<th>Description</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hang</td>
<td>Program became unresponsive OS terminated program</td>
<td>Crashed</td>
</tr>
<tr>
<td>OS-detected</td>
<td>ELZAR detected and corrected fault Fault did not affect output</td>
<td>Correct</td>
</tr>
<tr>
<td>SDC</td>
<td>Silent data corruption in output</td>
<td>Corrupted</td>
</tr>
</tbody>
</table>

**TABLE I: Fault injection outcomes classified.**
III. Evaluation

In this section, we answer the following questions:

- What is the performance overhead incurred by ELZAR, and what are the causes for high overheads (§III-B)?
- How many faults are detected and corrected by ELZAR during fault injection experiments (§III-C)?
- How does ELZAR perform compared to a state-of-the-art ILR implementation (§III-D)?

A. Experimental Setup

Applications. ELZAR was evaluated on two benchmark suites: Phoenix 2.0 [12] and PARSEC 3.0 [13]. Results are reported for all 7 Phoenix benchmarks and 7 out of 13 PARSEC benchmarks. The remaining 6 benchmarks from the PARSEC suite were not evaluated for the following reasons: bodytrack and raytrace use C++ exceptions not supported by ELZAR, facesim crashes with a runtime error when built with LLVM, fargmtine is based on OpenMP and does not compile under our version of LLVM, canneal has inline assembly and vips has long-double floats not supported by ELZAR.

All applications were built with LLVM 3.7.0 and ELZAR as described in §II-E. The native versions were built with msse4.2 and mavin2 flags to enable SIMD vectorization. The ELZAR versions were built with all vectorization disabled, i.e., with no-sse, no-avx, fno-vectorize, and fno-slp-vectorize flags. For all versions, all other compiler optimizations were enabled (O3 flag). Additionally, we used the fno-builtit flag to transparently link against our versions of libc and libm.

Note that we compare ELZAR against the native version with all AVX optimizations enabled. However, string match built with AVX shows a 60% increase in performance. Therefore, we decided to also show how ELZAR performs in comparison to the native version with AVX optimizations disabled; we refer to this experiment as smatch-na (for “string match no AVX”).

Datasets. For the performance evaluation, we use the largest available datasets provided by Phoenix and PARSEC. However, for the fault injection experiments, we use the smallest available inputs due to the extremely slow fault injection runs.

Testbed. The performance evaluation was done on a machine with two 14-cores Intel Xeon processors operating at 2.0 GHz (Intel Haswell microarchitecture3) 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. For performance measurements, we report an average of 10 runs.

For fault injections, we used a cluster of 25 machines to parallelize the experiments. We injected a total of 2,500 faults in each program. All programs-under-test were run with two threads to account for the impact of multithreading.

B. Performance Evaluation

Impact of ELZAR and scalability. The performance overheads incurred by ELZAR are shown in Figure 8. There is significant variability in behavior across benchmarks, with some showing overheads as low as 10% (matrix multiplication) and some exhibiting up to 20× worse performance (string match). On average, the normalized runtime of ELZAR is 4.1–5.6× depending on the number of threads.

For some benchmarks, there is also variability across the number of threads. Ideally, if a program has linear scalability, ELZAR should incur exactly the same performance overhead with any number of threads, e.g., as in case of word count or ferret. However, some benchmarks such as dedup are well-known to have poor scalability, i.e., with many threads they spend a lot of time on synchronization [14]. Thus, ELZAR’s overhead is partially amortized by the sub-linear scalability of these benchmarks.

To gain better understanding on the causes of high overheads as well as the causes of high variability across benchmarks, we gathered runtime statistics for native and ELZAR versions of all benchmarks. The results are shown in Tables II and III. The

3We also performed experiments on Intel Skylake but the results were similar to Intel Haswell. Therefore, we omit them in our evaluation.
TABLE II: Runtime statistics for native versions of benchmarks with 16 threads: L1D-cache and branch miss ratios, and fraction of loads, stores, and branches over executed instructions (all numbers in percent).

<table>
<thead>
<tr>
<th>Bench</th>
<th>L1-miss</th>
<th>br-miss</th>
<th>loads</th>
<th>stores</th>
<th>branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>hist</td>
<td>0.66</td>
<td>0.01</td>
<td>53.21</td>
<td>26.07</td>
<td>9.56</td>
</tr>
<tr>
<td>km</td>
<td>1.48</td>
<td>0.33</td>
<td>20.83</td>
<td>0.48</td>
<td>14.96</td>
</tr>
<tr>
<td>linreg</td>
<td>2.05</td>
<td>0.01</td>
<td>18.02</td>
<td>0.21</td>
<td>3.82</td>
</tr>
<tr>
<td>mmul</td>
<td>62.39</td>
<td>0.14</td>
<td>40.16</td>
<td>0.07</td>
<td>10.10</td>
</tr>
<tr>
<td>pca</td>
<td>12.19</td>
<td>0.27</td>
<td>14.21</td>
<td>0.21</td>
<td>3.79</td>
</tr>
<tr>
<td>smatch</td>
<td>0.12</td>
<td>0.70</td>
<td>11.61</td>
<td>14.35</td>
<td>22.40</td>
</tr>
<tr>
<td>black</td>
<td>10.94</td>
<td>3.31</td>
<td>29.75</td>
<td>23.63</td>
<td>13.67</td>
</tr>
<tr>
<td>fluid</td>
<td>0.40</td>
<td>1.21</td>
<td>9.38</td>
<td>2.84</td>
<td>15.63</td>
</tr>
<tr>
<td>ferret</td>
<td>4.30</td>
<td>3.80</td>
<td>30.08</td>
<td>13.55</td>
<td>12.01</td>
</tr>
<tr>
<td>scluster</td>
<td>1.69</td>
<td>12.65</td>
<td>11.77</td>
<td>25.86</td>
<td>14.29</td>
</tr>
<tr>
<td>swap</td>
<td>0.85</td>
<td>0.97</td>
<td>30.98</td>
<td>4.80</td>
<td>11.05</td>
</tr>
<tr>
<td>x264</td>
<td>0.34</td>
<td>0.31</td>
<td>26.83</td>
<td>8.32</td>
<td>21.00</td>
</tr>
</tbody>
</table>

benchmarks were run with 16 threads (and in the case of ELZAR, with all checks enabled) and profiled using perf-stat to collect hardware counters of raw events such as the number of loads, stores, branches, all instructions and AVX instructions only, etc.

Based on the information from Tables II and III, we can highlight several causes of high performance overheads. Firstly, as Table III shows, ELZAR leads to an increase in the total number of executed instructions of 4–8× on average. This disappointing high number is explained by the fact that ELZAR adds wrapper instructions for loads, stores, and branches, as well as expensive checks on synchronization instructions (see §II-D).

Second, looking at the achieved Instruction-Level Parallelism (ILP) in Table III, we notice that current x86 CPUs provide much better parallelization for regular instructions as compared to AVX instructions. As one example, linear regression achieves a high ILP of 6.51 instructions/cycle in native execution, but the AVX-based version reaches only a disappointing ILP of 1.7. Combined with the 10.49× increase in number of instructions for the AVX-based version, it is no surprise that linear regression exhibits an overhead of ∼5–8×.

Two benchmarks that show the lowest overheads are matrix multiplication and blackscholes. In the case of matrix multiplication, almost all of ELZAR’s overhead is amortized by a very poor memory access pattern that leads to 62.39% of all memory references missing L1 cache; in other words, matrix multiplication spends more time in waiting for memory than in actual computation. In the case of blackscholes, the main cause for low overheads is the small fraction of loads/stores (12.22%) and branches (15.63%).

Finally, we inspected the causes for extremely high overheads in string match. First of all, string match by itself significantly benefits from AVX vectorization based on our experiments. Indeed, ELZAR is ∼15–20× slower than the native version, but ∼10–14× slower than native with AVX vectorization disabled. Second of all, ELZAR increases the total number of executed instructions by a factor of 32. Upon examining the source code of string match, we noticed that it spends most of the time in bzero to nullify some chunks of memory. LLVM produces a very effective assembly for this helper routine, but ELZAR inserts wrappers and checks for the store and branch instructions in bzero, leading to much longer and slower assembly code.

Impact of checks. We also investigated the impact of checks inserted by ELZAR (see §II-D). Figure 9 shows the results of successfully disabling checks on loads, stores, branches, and all other instructions (e.g., function calls, function returns, atomics). Note that the results are shown for benchmarks run with 16 threads.

We observe that checks constitute a significant part of the overall performance overhead of ELZAR. For example, disabling checks on loads and stores decreases the overhead from 4.2 to 2.7× on average, a difference of 55%. Disabling checks on branches leads to a negligible overhead reduction of 4%, which proves that our branch checking scheme is very efficient (§II-D).

We also observe that disabling checks on loads and stores respectively reduces the overhead by 11% and 40%, i.e., checks on stores have higher overheads than checks on loads. The reason is that stores require to check both the address and the value to store whereas loads only need to check the address.

Floating point-only protection. As AVX was initially developed to accelerate floating-point calculations, it is interesting to study the overheads when applying ELZAR only to floating-point data. We thus developed a stripped-down version of ELZAR that replicates floats and doubles but not integers and pointers, and ran tests on several PARSEC benchmarks that contain sufficiently many floating-point instructions. Indeed, E−LZAR shows an overhead of ∼15–20×, i.e., 32% for blackscholes (47% of all instructions are floating-point), fluidanimate (32%), and swaptions (34%) [13].

Our results prove that ELZAR hardens floating points with a low overhead. Depending on the number of threads, we observe a 9–35% performance overhead over native for blackscholes, 4 10–18% for fluidanimate, and 40–60% for swaptions. The overhead is mainly caused by the checks on synchronization instructions.

C. Fault Injection Experiments

The results of the fault injection experiments are shown in Figure 11. On average, ELZAR reduces the SDC rate from 27% to 5% and the crash rate from 18% to 6%.

Histogram has the worst result with 12% SDC. It highlights ELZAR’s window of vulnerability: address extractions before loads and stores. If a fault occurs in the extracted address, it will be used to load a value from the wrong address, and this value will then be broadcast to all replicas. In other words, the fault will remain undetected and may lead to SDC (similarly, 4This is in line with the numbers reported by Chen et al. [15] where a single-threaded, manually written SSE-based version of blackscholes exhibits ∼30% overhead.

TABLE III: Runtime statistics for ELZAR and SWIFT-R versions of benchmarks with 16 threads: Instruction-Level Parallelism (ILP) and increase in the number of executed instructions W.R.T. native.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Instruction-Level Parallelism (ILP, instr/cycle)</th>
<th>Increase in # of instr w.r.t. native</th>
</tr>
</thead>
<tbody>
<tr>
<td>hist</td>
<td>1.39/2.13</td>
<td>8.56/6.17</td>
</tr>
<tr>
<td>km</td>
<td>3.48/2.58</td>
<td>6.37/4.34</td>
</tr>
<tr>
<td>linreg</td>
<td>6.51/1.70</td>
<td>10.49/4.33</td>
</tr>
<tr>
<td>mmul</td>
<td>0.22/0.96</td>
<td>4.47/7.77</td>
</tr>
<tr>
<td>pca</td>
<td>2.61/2.28</td>
<td>6.82/9.45</td>
</tr>
<tr>
<td>smatch</td>
<td>2.38/3.26</td>
<td>32.72/11.56</td>
</tr>
<tr>
<td>wc</td>
<td>1.31/1.77</td>
<td>6.14/3.42</td>
</tr>
<tr>
<td>fluid</td>
<td>1.83/1.77</td>
<td>1.70/5.18</td>
</tr>
<tr>
<td>ferret</td>
<td>1.04/1.75</td>
<td>4.64/3.68</td>
</tr>
<tr>
<td>scluster</td>
<td>1.11/1.81</td>
<td>4.32/6.33</td>
</tr>
<tr>
<td>swap</td>
<td>1.22/1.54</td>
<td>2.43/6.02</td>
</tr>
<tr>
<td>x264</td>
<td>0.68/1.22</td>
<td>3.77/3.87</td>
</tr>
<tr>
<td></td>
<td>1.97/2.06</td>
<td>3.50/4.40</td>
</tr>
<tr>
<td></td>
<td>2.11/2.00</td>
<td>3.26/3.71</td>
</tr>
</tbody>
</table>
We re-implemented SWIFT-R because its source code was not publicly available; we employed manual assembly inspection to make sure our implementation of SWIFT-R produces fast and correct code.

In general, SWIFT-R incurs lower overheads than ELZAR, 2.5× against 3.7× on average. Interestingly, ELZAR performs better in three benchmarks, namely kmeans, blackscholes, and fluidanimate. To understand the differences between these approaches, we also report runtime statistics of SWIFT-R (Table III).

We can draw two conclusions. First, SWIFT-R benefits from higher ILP, which is the key for its low performance overhead. As discussed before, ELZAR takes a different stance and replicates not instructions but data; that is why it exhibits lower ILP but still performs on par with SWIFT-R in many cases.

Second, SWIFT-R significantly increases the number of instructions, which hampers its performance. ELZAR has a smaller increase, proving our hypothesis that AVX-based ILR leads to less code blow-up. For example, ELZAR outperforms SWIFT-R on blackscholes and fluidanimate exactly for this reason: even though SWIFT-R’s ILP is almost 2× higher than ELZAR, SWIFT-R produces ∼2.5–3× more instructions.

At the same time, SWIFT-R significantly outperforms ELZAR in benchmarks that are dominated by memory accesses. In these cases, ELZAR inserts a plethora of checks and wrappers, which results in a much higher number of instructions compared to SWIFT-R. This is exemplified by histogram, string match, and word count.

IV. CASE STUDIES

In this section, we report our experience on applying ELZAR to three real-world applications: Memcached, SQLite3, and Apache.

**Memcached key-value store.** We evaluated Memcached v1.4.24 with all optimizations enabled, including atomic memory accesses. The evaluation was performed locally on the same Haswell machine used for other experiments, with 1–16 cores dedicated to the Memcached server and all other cores to the YCSB clients [16] for generating workload. We opted to show the local performance of Memcached because the performance in a distributed environment is limited by the network and not by the CPU.

Figure 10a shows the throughput of native and ELZAR versions of Memcached run with two extreme YCSB workloads: A (50% reads, 50% writes, Zipf distribution) and D (95% reads, 5% writes, latest distribution). We observe that ELZAR scales on par with native, achieving up to 72% of native throughput for workload A and up to 85% for workload D. We also observed in our experiments that the latency of ELZAR is ∼25% worse than native (not shown here). Such good results are explained partially by Memcached’s poor memory locality, which amortizes the costs of ELZAR.

**SQLite database.** We evaluated SQLite3 using an in-memory database and YCSB workloads, similar to Memcached. We should note that SQLite3 has a reverse scalability curve because it was designed to be thread-safe and not concurrent. Therefore, SQLite3 exhibits worse throughput with higher numbers of threads.
The performance results are shown in Figure 10b. ELZAR performs poorly, achieving only 20–30% of the throughput of the native version. This overhead comes from the high number of locally near loads and stores, as well as function calls and function pointers. In all these cases, ELZAR inserts additional checks and wrappers that significantly degrade performance.

Apache web server. We evaluated the Apache web server using its “worker multi-processing module” with a single running process and a varying number of worker threads. As a client, we used the classical ab benchmark which repeatedly requests a static 1MB web page.

Figure 10c shows the throughput with varying number of threads. ELZAR performs very well, with an average throughput of 85% compared to native. We attribute this good performance to the fact that Apache extensively uses third-party libraries that are not hardened by ELZAR.

\section{Discussion}

In this section, we highlight performance bottlenecks in the current AVX implementation that lead to high overheads of ELZAR.

\textbf{Loads, stores, and branches.} Even not taking into account the overhead of checks, ELZAR still performs 160% worse than the native version (see Figure 9, “all checks disabled”). This performance impact stems mainly from the three bottlenecks: loads, stores, and branches.

To understand the impact of each of the three main bottlenecks, we created a set of microbenchmarks. Each microbenchmark has two versions: one with the regular instruction (e.g., regular load) and one with the AVX-based instruction (e.g., AVX-based load as shown in Figure 4). In each microbenchmark, the instruction is replicated several times to saturate the CPU and wrapped in a loop to get execution time of at least 1 second.

The results of microbenchmarks are shown in Table IV. We conclude that adding extract-broadcast wrappers for AVX-based loads results in a $\sim2\times$ increase of load execution time. Similarly, adding ptest for AVX-based branches leads to an overhead of $\sim1.9\times$. Interestingly, AVX-based stores do not exhibit high overhead, which is explained by the fact that our Intel Haswell has only one port to process data stores and thus the store operation itself is a bottleneck even in the native version.

\textbf{Checks on loads and stores.} As can be seen from Figure 9, ELZAR’s checks on synchronization instructions contribute a significant amount of the overhead (30% on average). Specifically, checks on loads and stores account for most of the overhead because of the complicated sequence of check instructions (see Figure 6). At the same time, checks on branches add only 5% overhead due to an efficient re-use of ptest already needed for branching itself (see Figure 7).

\textbf{Missing instructions.} Our Intel Haswell supports the AVX2 instruction set. Though AVX2 provides instructions for almost all operations, some classes of operations are missing. Two prominent examples are integer division and integer truncation. In the case of integer divisions, ELZAR generates at least four regular division instructions and the corresponding wrappers to extract elements from the input YMM registers and insert elements in the output YMM register; with truncations, the situation is similar. Clearly, emulating such missing instructions via a long sequence of available AVX instructions can lead to tremendous slowdowns. For example, our microbenchmark for truncation exhibits overheads of $5\times$.

\section{Conclusion}

We presented ELZAR, an AVX-based implementation of Instruction-Level Replication (ILR). ELZAR achieves fault tolerance not by replicating instructions, but by replicating data inside AVX registers. To our disappointment, we found out that AVX suffers from several limitations that lead to poor performance when used for ILR. The observed performance bottlenecks are primarily caused by the lack of suitable control flow and memory access instructions in the AVX instruction set, which necessitates the introduction of wrappers and ineffective checks for some types of instructions.

More implementation details and experimental results as well as our proposed improvements for the future generations of AVX that can lower the overheads of ELZAR are presented in the accompanying technical report [17].

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Loads & Stores & Branches \\
\hline
average-case  & 1.96 & 1.00 & 1.86 \\
worst-case  & 2.06 & 1.14 & 1.89 \\
\hline
\end{tabular}
\caption{Normalized runtime of AVX-based versions of microbenchmarks w.r.t. native versions.}
\end{table}

\begin{thebibliography}{10}
\end{thebibliography}