Abstract—The systems resilience research community has developed methods to manually insert additional source-program level assertions to trap errors, and also devised tools to conduct fault injection studies for scalar program codes. In this work, we contribute the first vector oriented LLVM-level fault injector VULFI to help study the effects of faults in vector architectures that are of growing importance, especially for vectorizing loops. Using VULFI, we conduct a resiliency study of nine real-world vector benchmarks using Intel’s AVX and SSE extensions as the target vector instruction sets, and offer the first reported understanding of how faults affect vector instruction sets. We take this work further toward automating the insertion of resilience assertions during compilation. This is based on our observation that during intermediate (e.g., LLVM-level) code generation to handle full and partial vectorization, modern compilers exploit (and explicite in their code-documentation) critical invariants. These invariants are turned into error-checking code. We confirm the efficacy of these automatically inserted low-overhead error detectors for vectorized loops.

I. INTRODUCTION

The growing importance of resilience to bit-flip induced errors demands concerted and prompt action by the research community. Recent studies have shown that the soft error rates (SER) are expected to rise, and become one of the major causes for reducing the mean-time-to-failure (MTTF) in exascale systems [1], [2]. While additional or hardened hardware circuitry can prevent the onset of failures or detect them early in the stack, with the ever increasing push for power-efficient architectures, cross-layer resilience solutions that encompass all of the hardware and software stack are critically important to be developed. Given that resilience solutions typically incur a net drop in performance, it is critically important that the research community develop and make available a whole array of solutions that can then be evaluated over an adequate period of time by practitioners, so that the right combinations of solutions that minimize overheads may be chosen. It is also equally important for the research community to boost each others’ efforts by creating and sharing the necessary tooling support so that the solutions become available in a timely manner.

One of the central tools that most research groups bank on are bit-flip based fault injectors. These are tools that inexpensively and conveniently simulate bit-flips so that one may, using them, evaluate the efficacy of fault detection and correction mechanisms. There are many types of fault injectors: (1) direct beam studies, where chips and systems are subject to high-energy beams [3], (2) architectural level simulation platforms (e.g., SIMICS) where very detailed studies can be performed [4], (3) binary instruction level fault injectors (e.g., PIN-based injectors [5]), (4) LLVM-level fault injectors [6]–[8], and (5) other higher level fault injectors, including those that simulate faults at the source-code level [3].

While each fault injector type suits specific needs and while no one fault injector type subsumes another, there is increasing interest (as well as merit) in focusing on LLVM-based fault injectors: (1) these injectors are relatively easy to maintain, modify, and share; (2) LLVM [9], [10] offers increasing support for program analysis passes; (3) there is a significant growth in interest on part of the HPC community to adopt LLVM-based solutions, and (quite importantly for this paper) (4) LLVM extensions for supporting vector instruction types, and special hooks to pass information to downstream tools is on the ascent. Given all these points, our strong preference is to focus on LLVM-level fault injectors. In fact, we are one of the groups that has developed a fault injector called the Kontrollable Utah LLVM Fault Injector (KULFI, [6]). This injector has been put to use in several of our own projects [11], and by outside groups [12]. Calhoun et al. utilized KULFI to develop a new fault injector ‘FlipIt’ targeting MPI applications [13]. To our best knowledge, these fault injectors do not target vector instructions.

Encouraged by these successes, we set about to find out whether today’s resilience solutions are adequately developed to support exascale computing. We discovered many important areas requiring immediate attention:

(1) Encouraged by these successes, we set about to find out whether today’s resilience solutions are adequately developed to support exascale computing. We discovered many important areas requiring immediate attention:

(1) Even though vector architectures are central to gaining power efficiencies, none of the available fault injectors address these architectures. In order to support the research community in this area, we worked on not only making KULFI modular, but also equipping it with a rich array of features that support (or could be easily extended to support) multiple vector formats (including Intel’s SSE and AVX vector instruction set). The resulting fault injector ‘VULFI’1 (Vector oriented Utah LLVM Fault Injector) is our first main result.

(2) Vectorization is primarily supported in the form of: (i) language specific vector extensions which are supported in modern compilers such as GCC [14] and Clang [15], and (ii) dedicated programming languages with inbuilt support for data-level parallelism such as ISPC [16] and OpenCL [17] that can enable vectorization to occur more predictably and under programmer control. Thus, languages such as ISPC and

1VULFI is available at: http://formalverification.cs.utah.edu/fmr/vulfi
OpenCL, and their associated compilers must be part of the research focus; we found no prior work targeting this angle.

(3) There is the intriguing possibility that compiler writers of such languages have taken care to explain their code generator, and even provided some hints on the invariants being followed when specific situations are handled – for instance, partial vectorization supported by bit masking. We invested a significant amount of time studying ispc's publicly available code generator (well-supported in this effort by the compiler team who answered our questions promptly), and found that it offers another intriguing wrinkle in terms of fault detector synthesis: namely that these invariants could be turned into inexpensive error detectors. This achieves two purposes at once: (1) One may actually be able to exploit specific patterns during code generation to tune and generate low overhead error detectors. While no single error detector type is sufficient to trap all types of faults (and our detectors are no exception), the attraction of error detectors that incur low overheads, are effective at trapping many faults, and (last but not least) can be automatically generated and inserted is potentially of huge interest in transferring resilience research into practice. (2) Dialog between the resilience research community and the compiler community may be a two-way street in that knowing the needs of the resilience community, compiler writers may be encouraged to document their compiler-backends more and/or provide features that support the generation of even more low overhead resilience solutions.

Roadmap: In §II, we detail VULFI, capable of targeting instructions at LLVM’s intermediate representation (IR) level, and simulating soft errors occurring in a CPU’s vector units. Targeting vector instructions for fault injection requires the capability to distinguish between unmasked and masked vector instructions including architecture specific LLVM intrinsics². This distinction is crucial in deciding whether or not to target a particular vector lane for a fault injection. Also, given that a vector register consists of a fixed number of packed scalar registers, a systematic approach is developed to allow each of these scalar registers to be treated independently during fault injection.

In §III, we demonstrate extracting IR-level loop invariants for a foreach loop supported in the ISPC compiler in order to synthesize error detectors. Our findings highlight that the understanding of underlying code generation is central to discovering these invariants. We introduce our fault models, set up our definitions, and provide an overview of our case studies.

In §IV, we evaluate how well our detector types cover important situations in practice. We present a fault injection drive case study, done using VULFI, analyzing resiliency of nine diverse C++ and ISPC vector benchmarks. We also employ the IR-level loop invariants to build soft error detectors, reporting their efficacy and overhead, using Intel’s open source ISPC [16], [18] as the language and the compiler of choice.

²Intrinsics referred in this paper are listed in the Intrinsics.gen file distributed with the public release of LLVM 3.2.

II. VULFI: Fault Injector handling Vector Instructions

A. Terminology and Assumptions

We now define our terminology as well as some of our default assumptions.

Vector and scalar instructions: An LLVM IR instruction will be referred to as an “instruction.” A vector instruction has at least one vector type operand.³ A scalar instruction has no vector operand.

Vector and scalar registers: A vector register is an Lvalue register or a source operand of an instruction of vector type . A scalar register is an Lvalue register or source operand register of an instruction that has type integer, floating point, or pointer.

Vector length: The length of a vector register (V_l) is the number of scalar registers referred to within it.

getelementptr: At IR level, the address of an element of an aggregate data-structure, such as an array, is calculated using getelementptr instruction.

Vector instruction – extractelement: It extracts a scalar element from a given location of a vector register.

Vector instruction – insertelement: It inserts a scalar element at a given location of a vector register.

Intrinsics: An intrinsic in this paper refers to a special function whose implementation is provided by the LLVM compiler infrastructure. All LLVM intrinsics start with a prefix @llvm.

Code generation, Architecture: We refer to IR-level code generated by the ISPC compiler with -O3 optimization targeting x86.

B. Fault model

We consider a single-bit fault introduced at a random bit position of a CPU’s vector or scalar registers holding either integer or floating point values during operations such as: (1) moving values between registers, (2) arithmetic and logical operations, and (3) load/store operations. To provide coverage for these scenarios, we always target the Lvalue of an instruction with the exception of a store instruction which doesn’t have an Lvalue. This fault injection approach lets us simulate a variety of fault scenarios. For example, in the case of an arithmetic instruction, corrupting the Lvalue covers the scenarios where a bit-flip either occurs in one of the source operands of the instruction or in the arithmetic unit. For the case where a value is moved from a source register to a destination register, targeting the Lvalue covers both the scenarios where a bit-flip occurs either in the source register or in the destination register. Targeting the Lvalue of a load instruction covers the scenario where a bit-flip occurs in the load buffer. For a store instruction, we ensure that a value to be stored is considered for fault injection prior to the store operation.

³The data types referred to in this paper correspond to the type definitions provided in http://llvm.org/docs/LangRef.html
We inject exactly one fault during the whole execution span of a program. More specifically, for a given program executed under a given input, and having $N$ dynamic fault sites, exactly one fault site is selected at random with a uniform probability of $1/N$ for the fault injection. Here, a dynamic fault site refers to a fault site associated with a runtime instance of a static instruction. Similar fault models have been used in recent fault injection studies on scalar architectures [6], [19], [20]. VULFI uses this fault model for performing fault injections. It first builds a list of fault sites using the Lvalues of the target instructions. If an Lvalue is a vector register, then each of its scalar elements is considered a unique fault site. The target instructions are selected based on one of the fault site selection heuristics (§ II-C). Each fault site from the target list is then instrumented with a call instruction which invokes one of the VULFI’s runtime fault injection API functions.

C. Fault site selection

VULFI uses one of the following fault site selection heuristics to build an initial list of fault sites to be targeted for fault injections. Specifically, VULFI analyzes the forward slice of a fault site, to classify it into one of the following categories:

1) **Pure-data sites:** The forward slice of the fault site must not have any getelementptr (address calculation) or control-flow instructions.

2) **Control sites:** The forward slice must have at least one control-flow instruction.

3) **Address sites:** The forward slice must have at least one getelementptr.

For example, in Figure 3, a bit-flip occurring in the variable $i$ may cause the loop execution to either end prematurely or run greater than $n$ iterations. It may also become an out-of-bound index for the array $a[]$ thereby potentially causing an invalid memory reference. However, a bit-flip occurring in the variable $s$ will never affect the loop control neither will it cause an invalid memory reference. Therefore, the variable $i$ is an example of both a control site and an address site whereas the variable $s$ is an example of pure-data site. Figure 2 more formally shows how these three fault site categories relate.

D. Instrumentation workflow

Figure 4 shows the instrumentation workflow of VULFI using an example of a vector register of length four with each of its elements considered a unique fault site. VULFI performs following three key operations as part of the instrumentation process: (1) Iterates over each of the scalar element in the cloned value of the vector register; (2) In each step, VULFI extracts an uninstrumented scalar element (represented by a white circle), performs instrumentation, and inserts the result (represented as a solid black circle) into the vector register;
(3) Finally, VULFI replaces the original vector register with its new cloned and instrumented version, redirecting all the users of the original vector register.

Figure 5(A) illustrates this on a masked vector load operation followed by a masked vector store operation. The vector load and store operations are done using the x86 intrinsics @llvm.x86.avx.maskload.ps.256 and @llvm.x86.avx.maskstore.ps.256, respectively. VULFI maintains an inbuilt list of x86 intrinsics, which classifies whether any given intrinsic performs a masked vector operation. In the current example, this information is used to ascertain that both the intrinsics use execution mask value of the %floatmask.i register to enable or disable load or store operations along the vector lanes. Figure 5(B) shows the instrumented version of the vector load and store operations. Each scalar element of the vector register %0 (the chosen fault site for instrumentation) is extracted using extractelement instruction (locations L1 and L5). The respective execution mask values of the scalar elements is extracted from the vector register %floatmask.i (locations L2 and L6). An extracted element and its execution mask value is then passed on to the runtime fault injection API (injectFaultFloatTy() at location L7 in the current example) to perform actual fault injection at runtime.

III. ERROR DETECTORS USING LLVM IR-LEVEL INVARIANTS

We now describe two specific instances of how compilation methods can be exploited to generate error-checking invariants based on the code-generation logic of compilers.

A. Example 1: Loop Invariants in a foreach Loop Construct

An ISPC foreach loop accepts one or more dimension variables of integer types with the iteration space of each dimension variable bounded by an interval \([\text{start}, \text{end}]\). A foreach loop uses its dimension variables as iterators to iterate over the loop body. To maximize lane utilization, for a given dimension variable, ISPC uniformly distributes first \(\{n − (n \% V_l)\}\) loop iterations across \(V_l\) vector lanes, where \(n = \text{end} − \text{start}\). The rest of the \(n \% V_l\) loop iterations are handled separately.

Consider Figure 7 which presents the control-flow graph (CFG) of the vector copy function, vcopy_ispc of Figure 6. The uniform qualifier appearing in vcopy_ispc denotes that all vector lanes share the same address of arrays \(a1\) and \(a2\), as well as the variable \(n\). The foreach_full_body basic block executes \(\{n − (n \% V_l)\}\) times with all \(V_l\) vector lanes performing parallel copy operations. The remaining \(n \% V_l\) loop iterations are done in the basic block partial_inner_all_only. The values \(\{n − (n \% V_l)\}\) and \(n \% V_l\) are represented by the definitions aligned_end and nextras respectively in the entry basic block allocas. The definition new_counter is the loop iterator for the foreach_full_body basic block. Based on these facts, one can construct the following loop invariants for foreach constructs, as shown in Figure 8. Clearly, such invariants must always hold within, as well as upon exit from, foreach loop appearing in an ISPC program (to minimize overheads, we check them only upon exit).

B. Example 2: Protecting uniform variables

In ISPC, a uniform variable is shared across all vector lanes. The compiler achieves this by storing a uniform value first into a scalar register and then broadcasting it to a vector register. In Figure 9 (typical result of compiling a code block containing a uniform variable), uval is a scalar register storing a uniform value. This value is first copied to the first element of the vector register uval_broadcast_init, and subsequently broadcast to all other locations using shufflevector instruction. The resultant value is stored in the vector register uval_broadcast. A bit flip affecting any of the scalar elements of uval_broadcast can be detected by inserting a piece of checker code which ensures that all scalar elements hold the same value before every read from uval_broadcast (inexpensively achieved by XORing.) Such detectors can provide good, though not perfect, fault detection coverage at very low cost.

We have implemented a prototype LLVM transformation pass implementing the detector described in §III-A (implementing the detector described in §III-B will be part of our future work). Our prototype implementation automatically inserts a detector basic block for each occurrence of foreach loop in a program (Figure 8 highlights this block, namely foreach_fullbody_check_invariants). This block contains a call instruction which calls our runtime detector API that takes new_counter, aligned_end, and \(V_l\) as arguments. As noted earlier, we invoke the detector block only upon loop exit. Our preliminary results (discussed in §IV) are quite encouraging.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

The experiments described in the paper were carried out on an Intel’s Core™i7 4770 system running 64-bit Ubuntu 12.04 operating system and with 16GB of main memory. The VULFI development is done with LLVM version 3.2 and the ISPC benchmark programs are compiled using ISPC compiler version 1.8.1.

B. Execution strategy

A fault injection experiment involves executing a benchmark program twice using a randomly selected program input chosen from a predefined set of inputs. During the first execution, no faults are injected, the execution output is recorded, and a dynamic fault site is chosen at random from a list of dynamic fault sites. The dynamic fault sites list is built by selecting either pure-data sites, control sites, or address sites. The second execution involves actual fault injection into the dynamic fault site chosen during the earlier run, using the fault
A vector of length 4

User 1  ...  User N

Fig. 4: VULFI instrumentation workflow

model described in the § II-B. VULFI classifies the result of a fault injection experiment into one of the following categories by comparing the execution outputs from the two program executions:

1) **Silent Data Corruption (SDC)**: When the result of the faulty execution differs from that of the fault-free execution.

2) **Benign**: When no difference is observed between the executions.

3) **Crash**: When the faulty execution results in a system failure, a program crash, or any other issue that could easily be detected by the end user.

### C. Benchmarks

Table I lists 9 benchmarks that we use for our fault injection experiments using VULFI. Benchmarks fluidanimate and swaptions are drawn from the PARVEC benchmark suite [21], and are vectorized implementations of their respective serial versions available with the PARSEC benchmark suite [22], [23]. Benchmarks Blackscholes, Sorting, Stencils, and Ray tracing, are selected from the list of benchmarks available with the public release of the ISPC compiler. The remaining three benchmarks are our own ISPC implementations of the respective C++ versions made available as part of scientific computing library (SCL) by Burkardt [24]. For our fault injection studies, we target the fault sites corresponding to each vectorized function of these benchmark programs. The fault sites are selected using one of the heuristics explained in §II-C. The benchmarks are evaluated using AVX and SSE4 as target x86 vector instruction sets.

**Benchmark Characteristics**: Figure 10 shows the mix of scalar and vector instructions for all 9 benchmarks. A significant portion of instructions under pure-data and control fault site categories are vector instructions. Specifically, the number of vector instructions, averaged across all 9 benchmarks, stands at 67% and 43% for pure-data and control fault site categories. A seemingly low percentage of vector instructions under address category should be taken with a grain of salt because at the IR-level, a scalar address is frequently cast into a vector address as and when required to be used by a vector instruction. These details clearly highlight the importance of having a fault injector that can target vector instructions—something we achieved by creating VULFI.

### D. Fault Injection Study

Table I summarizes the average number of dynamic instructions observed for each of our benchmarks. In fact, the average dynamic instruction count easily runs into millions for most of the benchmarks. Given that fault-injection based studies require huge numbers of runs on top of such high dynamic instruction counts, a key goal is to minimize the overall computational effort by employing statistical measures, but still maintaining a high degree of confidence in the reported results. To this end, our fault injection study is done by performing statistically significant fault injection campaigns for each of the benchmark programs.

A fault injection campaign comprises 100 independent fault injection experiments. The SDC rate calculated for a fault injection campaign is considered a unique random sample. We run a sufficient number of fault injection campaigns until: (1) the sample distribution becomes normal or near normal, and (2) for a target confidence level of 95%, the margin of error for the distribution falls within the range of ±3%. We observe that for each of our benchmarks, running 20 fault injection
Due to the (relatively) smaller size of these benchmarks, we follow a more comprehensive evaluation strategy by carrying out 2000 fault injection experiments for each of the micro-benchmarks under each of the fault site categories pure-data, control, and address. The detector’s effectiveness is measured in terms of percentage of fault injection experiments that end up in SDCs, together with the number of SDCs that get flagged by our detectors.

The loop invariants in Figure 8 depend on the IR-level loop iterator new_counter. The value of this loop iterator is used to evaluate the loop exit condition, and also to calculate the addresses of the array elements referenced in the micro-benchmarks. Therefore, fault sites affecting the loop iterator will be categorized as either a control site or an address site or both but can never be a pure-data site adhering to the relation shown in Figure 2.

Figure 12 confirms our hypothesis, showing that no SDCs are detected when pure-data sites are targeted for fault injection. In contrast, faults affecting control sites lead to the highest SDC rates, namely 96.5% for vector sum. In addition, 48.7% of the total fault injection experiments that end up in SDCs are also successfully detected.

Overall, the highest SDC detection rate is witnessed under control site category, with detectors approaching a detection rate of 57% for both vector copy and dot product micro-benchmarks. Faults affecting address sites report a relatively low SDC rate. This is because a substantial number of fault injection experiments end up in program crashes.

E. Error Detection Study

We evaluate the efficacy of the error detectors based on the (relatively) smaller size of these benchmarks, we follow a more comprehensive evaluation strategy by carrying out 2000 fault injection experiments for each of the micro-benchmarks under each of the fault site categories pure-data, control, and address. The detector’s effectiveness is measured in terms of percentage of fault injection experiments that end up in SDCs, together with the number of SDCs that get flagged by our detectors.

The loop invariants in Figure 8 depend on the IR-level loop iterator new_counter. The value of this loop iterator is used to evaluate the loop exit condition, and also to calculate the addresses of the array elements referenced in the micro-benchmarks. Therefore, fault sites affecting the loop iterator will be categorized as either a control site or an address site or both but can never be a pure-data site adhering to the relation shown in Figure 2.

Figure 12 confirms our hypothesis, showing that no SDCs are detected when pure-data sites are targeted for fault injection. In contrast, faults affecting control sites lead to the highest SDC rates, namely 96.5% for vector sum. In addition, 48.7% of the total fault injection experiments that end up in SDCs are also successfully detected.

Overall, the highest SDC detection rate is witnessed under control site category, with detectors approaching a detection rate of 57% for both vector copy and dot product micro-benchmarks. Faults affecting address sites report a relatively low SDC rate. This is because a substantial number of fault injection experiments end up in program crashes.

---

**Fig. 5:** Uninstrumented and instrumented (A,B) masked vector load and store instrns.
void vcopy_ispc(uniform int a1[], uniform int a2[], uniform int n){
    foreach (i = 0 ... n){
        a2[i]=a1[i];
    }
    return;
}

Fig. 6: ISPC implementation of vector copy

Fig. 7: Control flow graph of the vcopy_ispc() function with a detector block inserted

Invariant 1: new_counter ≥ 0
Invariant 2: new_counter ≤ aligned_end
Invariant 3: (new_counter % Vl) == 0

Fig. 8: Loop invariants for foreach_full_body basic block

Fig. 9: Broadcasting the value of the uniform variable uval to a vector register
Fig. 10: Composition of vector and scalar instructions in the benchmark programs

Fig. 11: Result of fault injection experiments for the vector benchmark programs

Fig. 12: SDC detection rate on micro-benchmarks using the invariants based detectors
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program</th>
<th>Language</th>
<th>Test Input</th>
<th>Target</th>
<th>Average Dynamic Instruction Count (in millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parvec</td>
<td>Fluidanimate</td>
<td>C++</td>
<td><code>sim_small</code> <code>sim_medium</code></td>
<td>AVX</td>
<td>170.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>sim_medium</code> <code>sim_large</code></td>
<td>SSE</td>
<td>199.7</td>
</tr>
<tr>
<td>Swaptions</td>
<td></td>
<td>C++</td>
<td>Swaptions: [16,64] Simulations: [100,200]</td>
<td>AVX</td>
<td>19.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>16.0</td>
</tr>
<tr>
<td>Blackscholes</td>
<td></td>
<td>ISPC</td>
<td><code>sim_small</code> <code>sim_medium</code> <code>sim_large</code></td>
<td>AVX</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>1.9</td>
</tr>
<tr>
<td>ISPC</td>
<td>Sorting</td>
<td>ISPC</td>
<td>1D Array length: [1000,100000]</td>
<td>AVX</td>
<td>5.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>5.4</td>
</tr>
<tr>
<td>Stencil</td>
<td></td>
<td>ISPC</td>
<td>2D Array dimension: Min. – 16x16 Max. – 64x64</td>
<td>AVX</td>
<td>57.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>69.3</td>
</tr>
<tr>
<td>Ray tracing</td>
<td></td>
<td>ISPC</td>
<td>Camera input: Sponza, Teapot, Cornell</td>
<td>AVX</td>
<td>69.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>68.8</td>
</tr>
<tr>
<td>SCL</td>
<td>Chebyshev</td>
<td>ISPC</td>
<td>Degree: [1,256]</td>
<td>AVX</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>Jacobi</td>
<td>ISPC</td>
<td>2D Array dimension: Min. – 32x32 Max. – 192x192</td>
<td>AVX</td>
<td>52.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>Conjugate Gradient</td>
<td>ISPC</td>
<td>2D Array dimension: Min. – 32x32 Max. – 256x256</td>
<td>AVX</td>
<td>45.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSE</td>
<td>43.6</td>
</tr>
</tbody>
</table>

TABLE I: List of Benchmarks used in the Fault Injection Study

The overhead incurred by our detectors is measured by executing and comparing the runtimes of an instrumented program binary with and without the detector block inserted. Average overhead is calculated by averaging the overhead data from 2,000 individual runs for each micro-benchmark. A low average overhead of approximately 8% is witnessed across all three micro-benchmarks. We believe that with increasing operation counts inside the `foreach` loop body (compared to our very short loop bodies), the overhead introduced by these detector blocks will further get amortized.

The examples presented in §III and the preliminary results discussed here have been quite encouraging, and we believe that we have barely scratched the possibility-space of exploiting compilation-aware detectors. This line of work, especially targeting vector instruction sets and SPMD languages, will be the main focus of our future work.

V. RELATED WORK

We have discussed pertinent related work throughout the paper. Here we present additional related research efforts.

A recent fault injection study done by Hari et al. introduces an assembly-level fault injector SASSIFI built specifically for NVIDIA’s CUDA architecture [26]. Another CUDA-centric fault injection study done by Fang et al. introduces a new fault injector GPU-Qin [27] which uses the CUDA debugger tool to insert breakpoints at program locations where fault injections are to be done. In contrast to these fault injectors, VULFI targets vector instructions at LLVM-level with the awareness for architecture specific vector extensions such as Intel’s AVX and SSE instruction sets. We believe that by targeting LLVM-level vector instructions enables VULFI to support any SPMD front-end compiler which uses LLVM as the back-end for machine code generation.

VI. CONCLUSION & FUTURE WORK

Despite the flurry of research underway in system resilience, very few solutions (including our past contributions) have been adopted and put into practice. The “sticker shock” of suffering a flat-out \(~25\%\sim 2\times\) overhead (typical figures for various error detectors and dual modular redundancy) can be unpalatable; a practitioner might prefer going back to an older lithography, suffering less errors (and overhead). The burden of manually inserting detectors into the source code can also hinder adoption. Finally, the non-availability of detector-types (and even the means for conducting studies) pertaining to vector instruction sets and SPMD languages further hinders adoption.

Our primary contribution in this paper is a well-engineered fault injector (VULFI) catering to systematic fault injection studies for vector instruction architectures. While we cannot say that our overheads are still within the ballpark of unquestioned acceptance, the detectors prove to be surprisingly lightweight, can be automatically generated and inserted during compilation, and may, in the grand scheme of things, provide the right kind and level of solution. Given that this is the first (as far as we know) vector-oriented fault injection framework and first attempt to exploit the logic of compilation to assist with system resilience, but also given that we now see a huge array of possibilities to expand our initial studies in our future work, we believe that this direction has a very good chance of yielding artifacts that will transition into practice.

VII. ACKNOWLEDGEMENT

This work was supported in part by the U.S. Department of Energys (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award 66905. Pacific
Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830. The Utah authors were supported under the same DOE project with award number 55800790, and also NSF Award CCF 1255776. We would also like to thank the developers of Intel’s ISPC compiler, especially Matt Pharr and Dmitry Babokin, for their great support in fixing a bug and promptly answering our queries.

REFERENCES


