CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

SIMD-Based Soft Error Detection

Zhi Chen, Alexandru Nicolau, Alexander V. Veidenbaum

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{zhic2, anicolau, aveidenb }@uci.edu

CECS Technical Report 15-04

November 17, 2015

SIMD-Based Soft Error Detection

Zhi Chen, Alexandru Nicolau, Alexander V. Veidenbaum
Department of Computer Science, Universiyt of California, Irvine
zhic2, anicolau, aveidenb@uci.edu

Abstract

Soft error rates in processors have been increasing with decreasing feature size and larger chips. Both hardware or software based solutions have been proposed to address this problem. However, this leads to significant overheads in chip area, performance, and/or energy. This paper proposes a novel, software-only, solution to the problem. It uses instruction duplication to detect and eventually correct transient faults with lower overhead than prior solutions. This is achieved by exploiting redundancy within SIMD instructions. The solution is implemented in the LLVM compiler. Execution of a set of compiled benchmarks shows that SIMD based instruction duplication introduces 12% and 9% performance and energy overheads, respectively, over the baseline. The same overheads become 21% and 14% when error checking and branching code is added.

Keywords Soft errors, fault tolerance, SIMD, vectorization

1. Introduction

Soft errors (SE) are transient errors in computer hardware caused by cosmic radiation, e.g. alpha particles and high-energy neutrinos. The impact of such a particle is of a very short duration, is very localized and happens in isolation, i.e. multiple simultaneous soft errors are very unlikely. Thus another term for such errors is Single Event Upsets. These errors can cause bits of storage elements to flip or disrupt the operation of a combinational logic circuit [29]. This may corrupt the output of an application or even crash a computer system. SRAM cells are very prone to such errors and have been enhanced via circuit design and error checking codes (ECC) to improve resilience. Latches and combinational logic have so far not been a problem, but their transient errors are also increasing [27]. Many-core chips[1] and large systems built from them will need to be protected from such errors.

Technology scaling with its reduced feature size and lower supply voltage provides processors better performance and energy efficiency. It also leads to variation in process, voltage, and temperature. The variation in turn makes computer systems more susceptible to soft errors, posing significant reliability challenges [17].

Previous work has proposed both hardware and software solutions to soft errors. Large SRAM arrays, e.g. caches, have high soft error rates (SER). This has been addressed via circuit techniques and through the use of error detection/correction codes. It was estimated in prior research that SERs for SRAM will remain roughly constant over several technology generations, at 10^{-4} FIT/bit [29, 34]. Combinational logic is more resistant to transient faults due to error masking, e.g. the fact that the output value of a logic circuit may not change even if there was a soft error. A circuit may also "recover" from an error before the result is latched. SERs for latches and combinational logic range from 10^{-5} to 10^{-3} FIT/bit or roughly 0.5 upsets per year per chip [28, 29]. These errors are thus hard to detect and, if detected, correct in hardware although future processors may have parity on all latches. Furthermore, the wide use of aggressive dynamic voltage scaling will further worsen the SER since it increases exponentially as voltage decreases [7].

The only hardware techniques that can detect and correct soft errors are based on hardware redundancy, e.g. Dual modular redundancy(DMR) and triple modular redundancy (TMR). However, DMR and TMR have significant chip area, performance and energy overheads. These costly approaches are only used for mission-critical applications, they are impractical for commodity processors where occasional errors are not a concern [12].

Software implemented fault tolerance is the only alternative for processors without full hardware fault tolerance. One such software solution is instruction duplication. Each instruction in a program is executed twice and the two results are compared. The probability that both have been affected by the same soft error is negligible and thus the comparison will detect a soft error. The duplication of instructions and addition of comparisons to check the outputs of the two instructions can be implemented in a compiler and thus fully automated. Instruction duplication schemes, such as EDDI [18] and SWIFT [23] , were developed utilizing instruction level parallelism (ILP) to overlap execution of original instructions and their duplicates. However, instruction duplication has a high performance overhead. EDDI reduced overhead of duplication by performing checks less frequently (i.e. not for every duplicated instruction). Still, the overhead remained significant.

Chen et al. [6] presented a feasibility study of a software solution to instruction duplication using vectorization. Instead of replicating an original instruction they replicated its operands using SIMD registers and performed an SIMD operation for duplicated execution. Their study focused on floating point computations in application kernels and manually inserted SSE/AVX intrinsics at the source level for duplication. It assumed that memory hierarchy was protected by ECC and did not duplicate memory accesses. Integer instructions were not considered. The results of the study indicated that the overhead of full duplication using SIMD instructions was quite low but checking every instruction was still expensive.

The goal of the work presented here is to automatically generate instruction duplication in a compiler. It focuses on soft error detection. Soft errors are infrequent enough to allow correction to be relatively slow. Detection, on the other hand, has to be continuous and thus fast. The duplication is accomplished by using SIMD vector instructions. SIMD instructions are available today in most commercial processors for both integer and floating point data types. They operate on separate, wider registers that fit multiple operands. This allows operands of the original instruction and its duplicate to be packed in the same SIMD register and instruction "duplication" is replaced by execution of a single SIMD instruction. Error checking consists of comparing the low and high words of the re-

[1] This work concentrates on soft error detection/correction within a single core and leaves multicores for future work.

sult register. This work also assumes that caches and memory are protected by the ECC.

Compiler instruction duplication has to address several issues. First, one needs to decide at what point in the compilation process to perform the duplication. If performed before optimizations are applied, the duplicated instructions will be dead code eliminated. It also cannot be performed after register allocation. The right place depends on a compiler infrastructure used. Additional registers needed for duplicated instructions increase register pressure which is another issue to be solved. This paper presents compilation for SIMD-based instruction duplication using the LLVM compiler infrastructure [13]. Duplication is performed at the intermediate representation (IR) level, after standard transformations and optimizations have been performed but before register allocation. This allows the register allocator to deal with original and duplicated instructions simultaneously. In fact, using SIMD registers allows us to duplicate operands without allocating additional registers. All integer and floating point instructions, except for branches/jumps, loads and stores are duplicated. In addition, checking code is added to compare results of a duplicated execution. Two versions of checking were implemented and compared in this work. Execution of compiled benchmarks, including some of the SPEC2000 and SPEC2006 codes, shows that SIMD-based duplication has a significantly lower overhead than prior approaches.

The contributions of this work are as follows:

- A compiler approach using SIMD vector units for instruction duplication and error checking is presented.
- A compiler implementation of the proposed approach is presented and discussed.
- Evaluation of benchmarks using full instruction duplication, including the library calls that can be supported by LLVM with vector compatible prototypes, was performed and showed that the approach has a relatively low overhead in performance, energy, and code size compared to prior work.

The remainder of this paper is organized as follows. Section 2 details the framework and the implementation of the proposed fault tolerance approach. Section 3 presents the experimental results of the performance, energy, and code size overheads caused by our technique. Section 4 gives an overview of the related work. Finally, we conclude the work and describe some future work in Section 5.

2. Compilation

This section describes our SIMD-based compilation approach to instruction duplication using the LLVM compiler infrastructure. To be specific, it assumes Intel SIMD instructions and uses both SSE and AVX2 instructions. Of course, the approach can be applied to other SIMD instruction sets. Examples below use floating point operations.

Figure 1 illustrates the basic idea of scalar and SIMD-based instruction duplication. The figure uses a source-level "instruction" for simplicity: $A[i] = B[i] + C[i]$. Figure 1(a) shows the scalar instruction duplication. It assumes that arrays A, B, and C are in memory. Two independent instructions are generated, $A[i] = B[i] + C[i]$ and $A'[i] = B[i] + C[i]$ using the same input operands loaded from memory. A comparison is inserted to check that $A[i]$ and $A'[i]$ are equal. A branch is also required to deal with an error. A recovery can be performed by re-executing the same instruction a third time and performing majority voting, or by using software checkpoints.

Figure 1(b) and Figure 1(c) demonstrate SIMD-based instruction duplication and error checking. Two possible cases are illustrated: code in Figure 1(b) has no SIMD instructions and code in Figure 1(c) has SSE instructions. In the former case, the operands

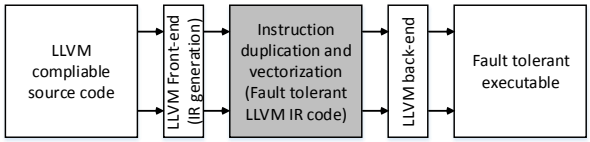


Figure 2: Compilation flow for our fault tolerant code generation approach.

$B[i]$ and $C[i]$ are loaded into the low quadword of two XMM registers. A broadcast instruction duplicates the value in low quadword to the high quadword (the dark grey boxes). An SSE addition is performed using the SIMD registers with duplicated operands. The results, $A[i]$ and $A'[i]$, are in the same XMM register after the computation. One only needs to check the equality of the words in the same register for error detection. However, the SSE (and AVX) instruction set does not contain an instruction for such a comparison. The result register thus needs to be copied with a shuffle to another register before the comparison.

Figure 1(c) shows the duplication in a code with SSE instructions (vector length=2). Duplication in this case requires a wider register, e.g. a YMM register (vector length=4). Otherwise the code is the same as the one shown in Figure 1(b).

Finally, a code containing SIMD instructions using the widest SIMD registers available on the target processor, e.g., a YMM register for AVX instructions in our case, cannot be duplicated in this manner. The scalar duplication approach in Figure 1(a) can be used for such instructions.

EDDI [18] duplicated data in memory for additional resilience and loaded input operands twice in the scalar duplication case. This work does not duplicate data in memory as this is too expensive and the memory hierarchy is protected by ECC. A register operand is also not duplicated, instead the same register is read by an instruction and its duplicate, assuming the register file is protected by hardware.

Overall, it is more appropriate to think of SIMD-based instruction duplication as data/operation duplication since only one SIMD instruction is sufficient for duplicated execution. This provides several advantages compared to the scalar version instruction duplication: reduced code size and register pressure, better performance, and easier compilation. However, additional instructions are needed to replicate the data in a vector register and to deal with the idiosyncrasies of SIMD instruction sets.

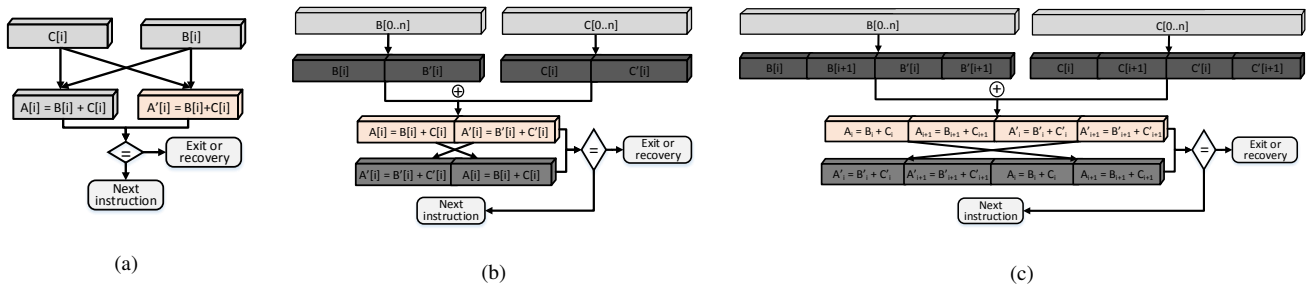
Next, an overview of the compilation framework is given in Section 2.1. Then, the use of conventional and our SIMD-based techniques to replicate instructions is presented in Section 2.2. Finally, different ways of inserting error checking code are discussed in Section 2.3.

2.1 Overall Compilation Process

Figure 2 shows the framework of the proposed solution using the LLVM compiler infrastructure [13]. The input to the framework is application source code in C/C++ [2]. The LLVM front-end compiles the source code and converts it into the LLVM *intermediate representation* (IR). A large set of standard transformations and optimizations are applied to the IR, producing an optimized LLVM IR form.

Our module, the grey box in Figure 2, starts with the optimized IR as the input and performs duplication in the IR code in a new

[2] or any other language that is supported by LLVM. For example, Dragonegg plugin is available for Fortran and Ada.



(a) (b) (c)

Figure 1: Scalar (a) and SIMD-based (b,c) source code duplication and error checking. (b): codes without SIMD instructions and (c): codes with SSE instructions.

LLVM pass. The LLVM IR uses a language-independent instruction set represented in a static single assignment (SSA) form. The new instruction duplication pass visits each individual IR instruction and analyzes it to determine if it is necessary to duplicate and vectorize this instruction. For instance, memory instructions and branches/jumps are not duplicated. The output of the pass is a modified IR form where an original instruction and its replica are "packed" in the same SIMD instruction. More details about the transformation will be given in the next section. A second new pass then is used to remove unnecessary instructions. Finally, another pass adds checking code for all duplicated instructions.

The transformed IR is the input to the (unmodified) LLVM backend which performs register allocation, additional optimizations, and code generation. The code generated in this work is for Intel processors with the AVX2 instruction set and can thus deal with both integer and floating point instructions. The proposed technique is implemented in the single threaded context, but we believe it can be easily extended to multithreaded codes.

2.2 Instruction Duplication

Let us first describe how different instruction classes are dealt with for soft error detection.

ALU instructions. These instructions are duplicated. The original instruction and its replica are "packed" in an equivalent SIMD instruction and their data is placed into SIMD registers.

Memory instructions. Stores are not duplicated.

Loads are also not duplicated, but a value is loaded into an SIMD register and then the value is broadcast to the rest of the lanes in the SIMD register.

Branches. Condition computation for a branch instruction is duplicated and checked. The PC update is not visible to user code and thus cannot be protected by instruction duplication. Existing software solutions based on run-time signature or assertions [4, 9, 19, 23, 31] or hardware solutions [5, 26] can be utilized, these are orthogonal to our work. Also note that branch target address computation is frequently not required when a BTB is used.

Function Calls. Library functions can be protected by duplication as computations. For instance, MKL and SVML support the vectorized forms of many library calls. One can vectorize these functions according to the input requirement of the LLVM front-end and leave the selection of SIMD intrinsic function calls to the back-end.

Function calls are viewed as synchronization points where the input parameters are checked before being passed to the callee. Note that only the parameters that are passed by value are protected. Parameters that are passed by reference are not duplicated because memory accesses are assumed to be reliable. This approach was also used in prior research, e.g. EDDI and SWIFT.

The scalar and the SIMD-based instruction duplication at the IR level are discussed next. An example, shown in Figure 3(a), will be used to illustrate the ideas. Figure 3(a) shows a code snippet from the *smvp* function of 183.equake benchmark. Figure 3(b) shows (partial) IR code produced by the LLVM front-end. The IR code contains both scalar and vector operations. The latter can be distinguished by "$\langle 2 \times double \rangle$" type.

2.2.1 Standard Instruction Duplication

Standard instruction duplication accesses each IR instruction and generates a duplicate instruction for each instruction class, as discussed above. This is the approach used in EDDI and SWIFT, except that a) in our case both scalar and vector instructions are duplicated and b) EDDI and SWIFT duplicated loads. The duplicate instruction is assigned a new (virtual) output register.

Figure 3(c) shows the IR code after the standard duplication is applied. The highlighted instructions (with underscore added to their output register) in the figure are the duplicates followed by the original instructions. For example, the multiplication and addition instructions, i.e., `%mul69`, `%add70`, `%Anext.1`, are duplicated as `%mul69_`, `%add70_`, `%Anext.1_`, respectively. The vector instructions, such as `%17`, `%23`, and `%24`, are handled the same way as the scalar instructions. Only instruction duplication is performed, the error checking code will be inserted in another pass.

2.2.2 SIMD-based Duplication

For SIMD-based duplication each instruction in the IR code is examined and duplication is based on the instruction's class and data type. In all cases, the original instruction is retained for the duration of this pass to simplify compilation. It will be removed in a separate pass.

(i) Scalar integer instructions use scalar registers. For example, instruction `%Anext.1` in Figure 3(b) will be allocated to a 32-bit scalar register. A scalar integer instruction will be changed to an SIMD instruction. For instance, Figure 4 shows that `%Anext.1` is replaced by its vector counterpart, `%Anext.1_`.

(ii) Scalar floating point instructions. Intel processors use SIMD registers for floating point operations by default, thus duplication of a scalar floating point instruction only requires value duplication. For example, `%mul69_` and `%add70_` in Figure 4 will use the same registers as used by `%mul69` and `%add70` in Figure 3(b), respectively. The technique used for these instructions corresponds to Figure 1(b). The solution for scalar integer instructions discussed above can be applied if the floating point operations do not use SIMD registers by default.

(iii) SSE instructions. Duplicating SSE instructions, such as `%17`, `%23`, and `%24` in Figure 3(b) requires wider registers, e.g. AVX2

```
for (i = 0; i < nodes; i++) {
  Anext = Aindex[i];
  Alast = Aindex[i + 1];

  sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + \
         A[Anext][0][2]*v[i][2];
  sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + \
         A[Anext][1][2]*v[i][2];
  sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + \
         A[Anext][2][2]*v[i][2];

  Anext++;
  ...
}
```

(a)

```
1  for.body: ; preds = %while.e, %ent
2    %iv1 = phi i64 [%iv.nxt2, %while.e], [0, %ent]
3    ...
4    %4 = load double* %3, align 8
5    ...
6    %13 = insertelement <2xdouble> undef, double %4, i32 0
7    ...
8    %17 = fmul <2 x double> %14, %16
9    ...
10   %23 = fmul <2 x double> %20, %22
11   %24 = fadd <2 x double> %17, %23
12   ...
13   %arridx65 = getelementptr double* %32, i64 1
14   %34 = load double* %arridx65, align 8
15   %mul69 = fmul double %8, %34
16   %add70 = fadd double %mul61, %mul69
17   ...
18   %Anext.1 = add i32 %0, 1
19   %cmp1 = icmp slt i32 %Anext.1, %1
20   br i1 %cmp1, label %while.b, label %while.e
```

(b)

```
for.body: ; preds = %while.e, %ent
%iv1_ = phi i64 [%iv2_, %while.e], [0, %ent]
%iv1 = phi i64 [%iv.nxt2, %while.e], [0, %ent]
...
%4 = load double* %3, align 8
...
%13_ = insertelement <2xdouble> undef, double %4, i32 0
%13 = insertelement <2xdouble> undef, double %4, i32 0
...
%17_ = fmul <2 x double> %16_, %14_
%17 = fmul <2 x double> %16, %14
...
%23_ = fmul <2 x double> %22_, %20_
%23 = fmul <2 x double> %22, %20
%24_ = fadd <2 x double> %17_, %23_
%24 = fadd <2 x double> %17, %23
...
%arridx65 = getelementptr double* %32, i64 1
%34 = load double* %arridx65, align 8
%mul69_ = fmul double %8, %34
%mul69 = fmul double %8, %34
%add70_ = fadd double %mul61_, %mul69_
%add70 = fadd double %mul61, %mul69
...
%Anext.1_ = add i32 %0, 1
%Anext.1 = add i32 %0, 1
%cmp1_ = icmp slt i32 %Anext.1_, %1
%cmp1 = icmp slt i32 %Anext.1, %1
br i1 %cmp1, label %while.b, label %while.e
```

(c)

Figure 3: Instruction duplication example. (a) Source code, (b) Partial LLVM bitcode, (c) Transformed LLVM bitcode. A highlighted IR instruction is a duplicate, followed by the original instruction.

registers. For example, instructions `%17_`, `%23_`, and `%24_` in Figure 4 correspond to the case in Figure 3(c).

(iv) AVX instructions. There are two viable solutions for this case: a) perform instruction duplication as in traditional approaches (shown in Figure 1(a), and b) half the loop unrolling factor so that SSE instructions will be generated. **We mainly focus on the later solution because it doesn't need to interfere the compiler optimizations for the fault tolerant code generation.**

A value is loaded once and broadcast to the unused lane, e.g. `%4_` in Figure 4. LLVM back-end generates *MOVDDUP* or *VMOVDDUP* for a floating point instruction depending on the type of the original instruction.

A $\langle 2 \times 1b \rangle$ result of a vector comparison needs to be converted to an SSE form. Otherwise, LLVM has to perform unpacking and packing for this comparison as the way it handles the instructions that cannot run in the SIMD manner, e.g. integer division (we will describe it in the next paragraph). This is done by sign extending the result to a $\langle 2 \times 64b \rangle$ vector instruction first. Then, a bitcast IR instruction is used to make the value compatible to an SIMD register (e.g. 128 bits for SSE). The comparison instruction, such as `%cmp1_` in Figure 4 is an example of this case. This forces LLVM to select vector comparison and branch instructions, e.g. *vpcmpeqq* and *vptest*, at the back-end automatically.

Most of the instructions in the original code become unnecessary after instruction duplication is completed. For example, most ALU instructions in Figure 3(b) can be removed when our SIMD-based instruction duplication successfully run on the code because each of them has a SIMD counterpart in the new generated IR code. However, some operations don't actually run in SIMD mode, e.g. *integer division* and *int32 to double conversion*, etc. LLVM will first unpack the two values from an SIMD register and then perform two scalar instructions sequentially on each value for these operations. Finally, the values produced by these two scalar instructions are packed into a vector register. More discussion about this case will be provided in the experimental section.

2.3 Checking Code Insertion

Error checking is implemented in a separate pass, (*checker insertion pass*), after instructions are duplicated and the unneeded instructions are removed. Checking can be performed in two different ways.

- Check after each duplicated instruction. The advantage of checking at this granularity is that error correction can be initiated immediately. The disadvantage is that it incurs a very high overhead.
- Check at certain program points such as before stores, function calls, conditional branches, etc. Overheads are thus signif-

```
for.body: ; preds = %while.e, %ent
  %Phi_ = phi <2 x i64> [%iv.nxt1_, %while.e], [zeroinitializer, %
      ent]
  ...
  %4 = load double* %3, align 8, !tbaa !16
  %ld4 = insertelement <2 x double> undef, double %4, i32 0
  %ld4_ = shufflevector <2 x double> %ld4, <2 x double> undef,
      <2 x i32> zeroinitializer
  ...
  %13 = insertelement <2 x double> undef, double %4, i32 0
  %13_ = shufflevector <2 x double> %ld4_, <2 x double> %ld4_,
      <4 x i32> <i32 0, i32 2, i32 1, i32 3>
  ...
  %17_ = fmul <4 x double> %14_, %16_
  ...
  %23_ = fmul <4 x double> %20_, %22_
  %24_ = fadd <4 x double> %17_, %23_
  ...
  %arridx65 = getelementptr double* %32, i64 1
  %34 = load double* %arridx65, align 8
  %ld34 = insertelement <2 x double> undef, double %34, i32 0
  %ld34_ = shufflevector <2 x double> %ld34, <2 x double>
      undef, <2 x i32> zeroinitializer
  %mul69_ = fmul <2 x double> %8_, %ld34_
  %add70_ = fadd <2 x double> %mul61_, %mul69_
  ...
  %Anext.1_ = add <2 x i32> %0_, <i32 1, i32 1>
  %cmp1_ = icmp slt <2 x i32> %Anext.1_, %1_
  %zext1_ = sext <2 x i1> %cmp1_ to <2 x i64>
  %bc1_ = bitcast <2 x i64> %zext1_ to i128
  %msk_1 = icmp ne i128 %bc_, 0
  br i1 %msk1_, label %while.b, label %while.e
```

Figure 4: LLVM IR after SIMD-based instruction duplication.

icantly reduced. However, recovery becomes harder as instructions may be committed and register state may be lost by the time the error is detected. This probably works best with checkpointing.

The way checking is performed in the second case depends on a program point it is inserted at. But the main idea is to only check store values before they can change program state. All other operations just propagate values to the stores.

- Stores. The duplicated store operand is checked.
- Conditional Branches. The condition evaluation result is checked.
- Calls. It is assumed that parameters are pushed onto the stack before the call. Thus they are stored to memory and will be checked. A parameter passed in a register needs to be checked separately.

For example, Figure 5 shows an example for different error checking techniques. Figure 5(a) represents a basic block containing 4 instructions. Assume instruction 4 is a store instruction for demonstration purpose (Generally, the last instruction in a LLVM IR basic block is a terminator instruction like return and branch, etc). Figure 5(b) is the basic block instrumented using our SIMD-based fault tolerant pass. The insertion of a checker has to divide a basic block into two smaller basic blocks as a branch instruction is inevitably introduced. Depending on the instruction mix of an application, a prohibitively large number of checkers might be required, as shown in Figure 5(c). Each checker needs at least 3 instructions, namely shuffle, comparison, and branch, therefore resulting in much more instructions.

Figure 5(d) shows the idea of the second solution to error checking where the operand of instruction 4 is validated before it is written to memory. Only one checker is needed for this case. The original basic block is only divided into 2 blocks when the checker is inserted before the store instruction. Therefore, much less instructions are introduced. The register pressure is also significantly reduced. The only disadvantage is that the immediate error correction is harder. For example, one might not be able to correct the fault happened in computing instruction 1 because instruction 3 might overwrite the registers used by instruction 1.

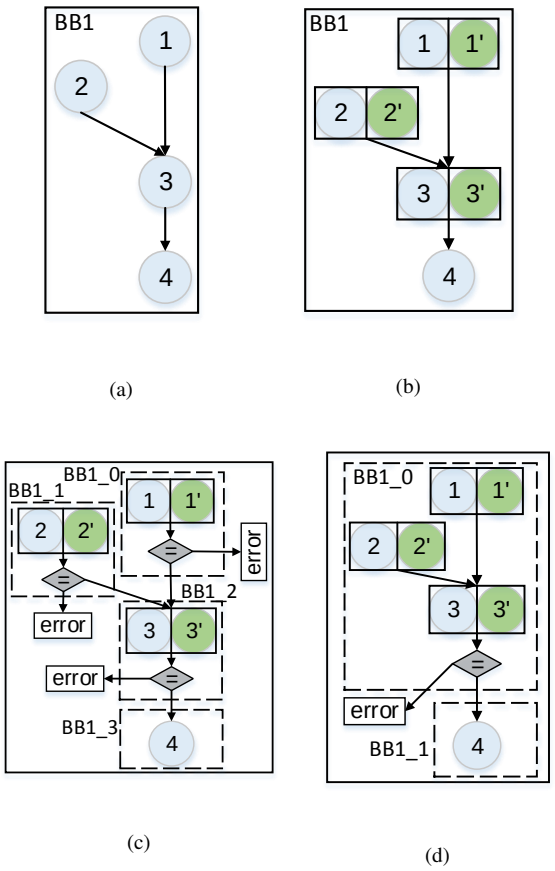


Figure 5: (a) A basic block consists of 4 instructions. (b) Transformed basic block using SIMD-based instruction duplication. Checkers are inserted: (c) after each duplicated instruction, (d) at the synchronization point.

3. Evaluation

This section provides experimental evaluation of the proposed error detection technique and compares its performance, energy, and code size with the baseline and prior duplication techniques. The next subsection describes our experimental setup and the characterization of benchmarks.

3.1 Experimental Setup

Compiled becnhmarks were executed on a system with an Intel Core i7-4770 processor and 8GB memory, running Linux 3.13.0 kernel. LLVM 3.4.2 was used (with -O3 flag enabled) to compile the original code and to add SIMD-based detection pass.

Benchmarks. Seven smaller benchmarks (IRSmk, Aobench, FFT, SpectralNorm, Blackscholes, Seidel-2D, and Linkpack), two benchmarks from SPECCFP2000 (179.art and 183.equake),

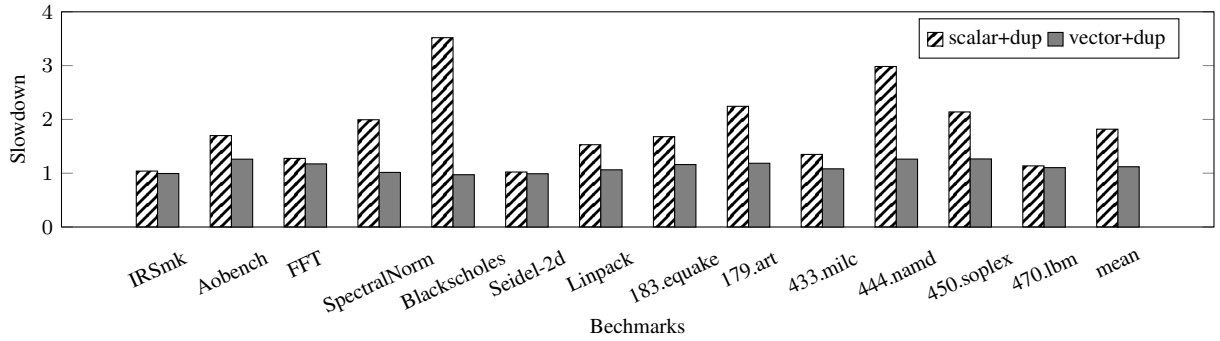


Figure 6: Performance slowdown of scalar and vector instruction duplication (no error checking).

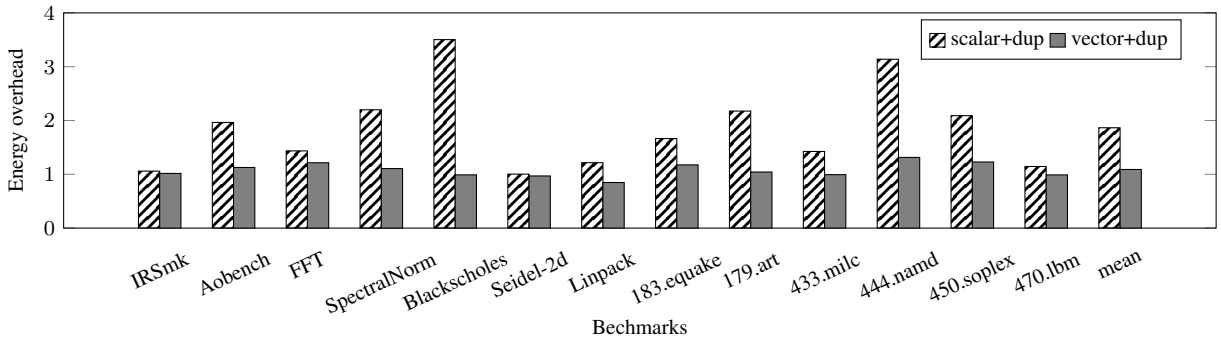


Figure 7: Energy overhead of scalar and vector instruction duplication (no error checking).

and four benchmarks from SPECCFP2006 (433.milc, 444.namd, 450.soplex, and 470.lbm) were used. The benchmarks were selected mainly because they are written in C/C++ and used both integer and f.p. instructions. All the SPEC benchmarks were run with reference data sets.

Hardware performance counters were used to measure the impact of the implemented approach on benchmark execution. The energy consumption results were collected using Likwid 3.1 [30]. The PKG counter was used to obtain energy consumed by the socket where the application was executed.

Multiple versions of each benchmark were compiled in: the original code (base), the code with SIMD-based instruction duplication only (vector+dup), SIMD-based instruction duplication with checkers inserted only at synchronization points (vector+dup+CS), and SIMD-based instruction duplication with vector error checking for each instruction (vector+dup+CE). The versions compiled using scalar instruction duplication and error checking were: pure scalar mode instruction duplication (scalar+dup), scalar mode instruction duplication with checking at synchronization points (scalar+dup+CS), and scalar mode instruction duplication and checking for each instruction (scalar+dup+CE). All results are normalized to the original program (base). "scalar+dup+CS" is similar to the approach implemented in SWIFT, except that the correctness was checked by duplicating comparison instead of using the signature-based technique.

3.2 Instruction Duplication Only

This section presents comparison of performance and energy consumption for pure instruction duplication (without error checking) for different versions of benchmarks.

Performance impact of pure instruction duplication. The performance slowdown of pure instruction duplication in scalar mode and vector mode is measured relative to the baseline. A program with vector mode instruction duplication is similar to the baseline program if the baseline program was not vectorized using the widest registers (YMM registers on the processor used for experiments). This is because only half of the lanes of an SIMD register are used for computation, the other half is used for duplication. Therefore, one can expect a minor performance slowdown by replacing scalar instructions with their vector equivalents. Figure 6 shows that the average slowdown for pure SIMD-based instruction duplication and scalar instruction duplication schemes are 1.12x and 1.82x, respectively. vector+dup technique is 37.9% faster, on average, than scalar+dup technique.

There are three major reasons for performance slowdown of vector mode instruction duplication.

First, a vector load needs to be "duplicated" by adding a broadcast of the low word to the high word. For example, we add *insertelement* and *shufflevector* instructions after each load in the IR code. The back-end generates a *vpbroadcastq* instruction after the *vmovd* instruction in assembly whenever we load an integer value from memory.

Second, although there are intrinsic functions provided in the front-end to support some vector library functions, the back-end doesn't generate vectorized library calls for some functions such as *sin* and *cos*, etc. Therefore, LLVM back-end has to unpack the values in a SIMD register and then perform two scalar library calls sequentially. Finally, these two results are packed back into a new SIMD register.

Finally, while AVX-2 supports integers, integer division and 32-bit integer to double precision floating point conversion are

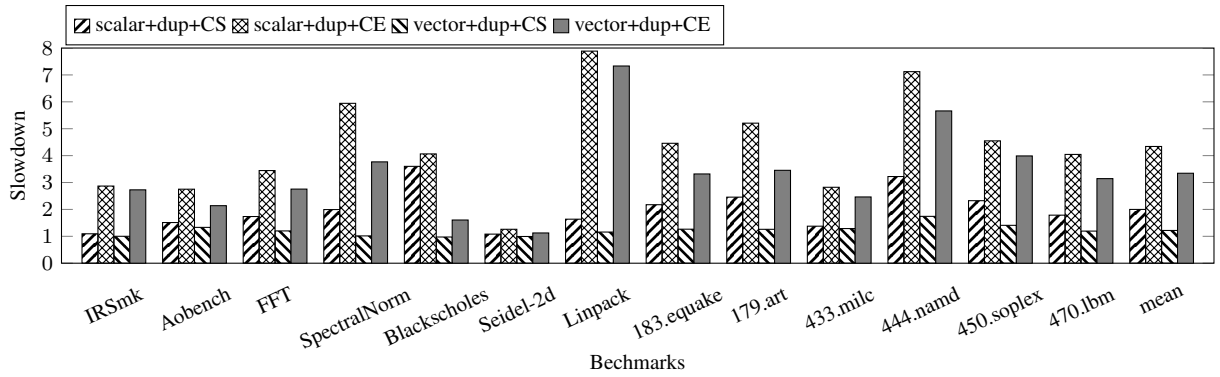


Figure 8: Performance slowdown for instruction duplication with selective and full error checking.

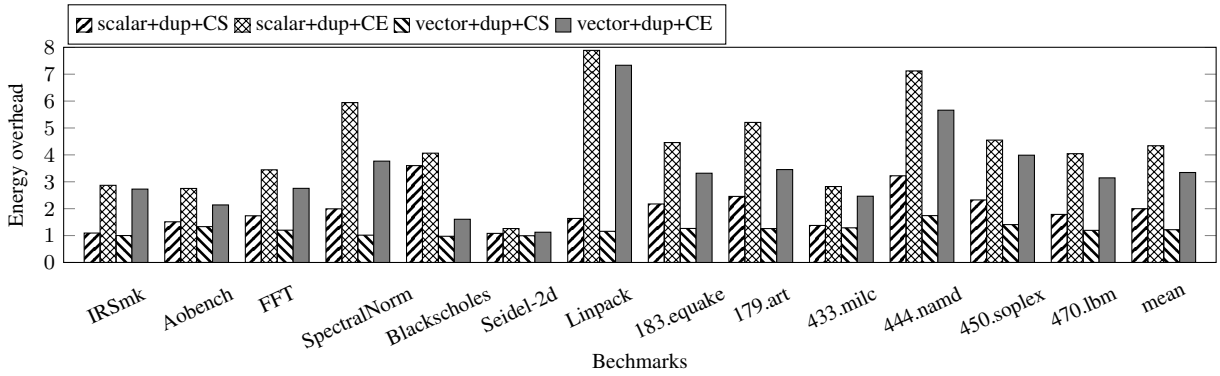


Figure 9: Energy overhead of instruction duplication with selective and full error checking.

currently not provided. These instructions require the unpacking and packing operations (as for library calls discussed above).

The impact of these depends on speficic benchmark.

- Memory intensive benchmarks, such as FFT and 183.equake, require many broadcast instructions to duplicate the load value which partially degrades the performance. There are also library calls (*sin* and *cos*) in the main function of these two benchmarks. Combining these two reasons, the slowdown for vector+dup of FFT and 183.equake is 1.17x and 1.16x, respectively.
- Aobench not only contains unvectorized intrinsic library calls, *sin*, *cos*, and *rand*, in one of the hot loops in the *ambient_occlusion* function but also has many integer to double precision floating point conversions, resulting in a 1.26x performance slowdown.
- Many integer division and type conversion instructions in 444.namd contribute to 1.26x performance slowdown.
- IRSmk, SpectralNorm, Blackscholes, 433.milc, and 470.lbm have lower performance slowdown because they don't require many additional vectorization instructions.

Figure 6 shows that pure scalar instruction duplication effectively reduces the performance by approximately a factor 2x for most benchmarks. Blackscholes surprisingly slows down by 3.5x for two reasons. First, around 2x instructions are executed compared to the baseline. Second, the register pressure increases dramatically since general registers are also used for instruction duplication. Therefore, much more spills are caused by scalar instruction duplication. However, the performance overhead for IRSmk and seidel-2d due to scalar instruction duplication is low because out-of-order instruction issue and ILP features of the processor overlap the execution of original instructions and their duplicates perfectly.

Energy consumption of pure instruction duplication. The energy consumption of scalar and vector mode instruction duplication is shown in Figure 7, normalized to energy consumed by the baseline. On average, pure scalar instruction and vector instruction duplication consumes up to 1.86x and 1.09x more energy, respectively. The energy consumption caused by vector mode instruction duplication is quite insignificant.

3.3 Instruction Duplication with Error Checking

This section presents results of duplication with error checking code insert in two different ways. Figures 8 and 9 show normalized execution time and energy consumption for different compiled versions of benchmarks with the error checkers inserted after each instruction (CE) or only at synchronization points (CS). It shows that scalar duplication+checking runs 1.99x slower and consumes 2.02x more energy, compared to the baseline, even when only checking for errors at synchronization points. Error checking after each duplicated instruction increases the performance and energy overheads sharply, to 4.34x and 4.58x on average, respectively.

The vector mode instruction duplication with error checking at synchronization points (vector+dup+CS) increases execution time by 21% and energy by 14%, on average, over the baseline. Compared to scalar+dup+CS, its performance and energy overheads are, on average, 39.2% and 43.6% lower, respectively, compared to scalar mode. The SIMD-based solution offers much better perfor-

Table 1: Standard deviation of performance and energy overheads.

Configuration	Performance	Energy
scalar+dup+CS	0.76	0.79
vector+dup+CS	0.21	0.16
scalar+dup+CE	1.85	1.97
vector+dup+CE	1.65	1.40

mance and energy efficiency compared to the scalar instruction duplication when checking all instructions. The average performance slowdown is 3.34x and the average energy consumption is 3.05x for vector+dup+CE.

One more observable benefit of our technique is that it delivers more stable performance and energy consumption across all benchmarks. Table 1 shows that standard deviation of performance slowdown for vector mode error detection is 0.21 and 1.65 for vector+dup+CS and vector+dup+CE, respectively. The standard deviations for the scalar error detection approach are 0.76 and 1.85, respectively. The standard deviations of energy consumption for scalar+dup+CS and scalar+dup+CE are 0.79 and 1.97, respecitvely. They are 0.16 and 1.40 for vector+dup+CS and vector+dup+CE schemes, respectively.

There are two reasons behind the above: a) while auto-vectorization does take place it leaves sufficient vector resources for redundancy, and b) not all of the benchmarks are able to benefit from the ILP capabilities of the processor for scalar fault tolerant techniques.

3.4 Code Size Overhead

Figure 10 shows the binary size of different compiled versions, normalized to the baseline with no instruction duplication. scalar+dup+CS and scalar+dup+CE duplicate all instructions in the IR form except stores, branches, function calls, and error checking code. vector+dup+CS and vector+dup+CE replicate the same type of instructions including the library calls that support SIMD intrinsic instructions in LLVM. scalar+dup+CS is 2.41x larger than the baseline which is consistent with results reported for SWIFT [23]. However, the proposed vector+dup+CS approach only increase the code size by an average of 1.90x. The binaries generated by scalar+dup+CE and vector+dup+CE (checking on every instruction) are, on average, 5.53x and 4.61x larger, respectively. Therefore, the vector mode error detection shrinks the binaries by an average of 20% and 17% compared to the scalar+dup+CS and scalar+dup+CE techniques, respectively. The major reason is because the duplicates are packed in the SIMD registers and executed in one vector instruction.

The proposed technique can further reduce overheads in terms of performance, energy and code size, if one can 1) vectorize all the library calls, 2) vectorize integer division and all integer to double precision floating point conversions, and 3) perform vector comparison for equality the low half bits and high half bits in a vector register. If there were SIMD instructions available for the first two cases, we could eliminate the packing and unpacking instructions to perform these operations.

4. Related Work

This paper focuses on using software instruction duplication with vectorization to reduce the soft error detection overheads in terms of performance, energy, and code sizes. The whole framework is implemented in the LLVM compiler without additional hardware modification. Previous work proposed hardware, software, and hybrid error detection techniques to address the soft error issue.

Hardware level. Two forms of redundancy are often exploited for error detection at this level, *structural redundancy* and *temporal redundancy*. DMR (dual-modular redundancy) and TMR (triple-modular redundancy) are structural redundancy based mechanisms to achieve virtually 100% error coverage by executing each operation on two or three exactly the same hardware units. However, these approaches are often deployed for mission-critical applications at a considerably high hardware cost. For example, targeting seven-nines (99.99999%) reliability, HP Integrity NS16000 and NS14000 servers provided DMR and TMR to tackle single hardware failures [3]. SIEMENS SIMATIC S7-400H offered redundant CPUs to protect them against failures [1]. DIVA[2] is a heterogeneous DMR alternative that uses a simplified core to monitor the operations performed by the more complex core. DIVA works well on large speculative RISC processors but less efficient on processors with little speculation. At finer granularity level, such as circuits, hardened circuits are designed for resiliency in high-radiation environments with the penalty of longer clock-cycle and about twice chip areas [15, 20].

AR-SMT [24], Simultaneous Redundant Multi-Threading (SRMT) [21], SRTR [32], CRT [16], and CRTR [10] are typical papers proposed to use temporal redundancy for transient error detection. These papers mainly focused on using redundant multithreading to detect soft errors. Two threads are always spawned to run an application. A leading thread is used to run the actual program and a trailing thread is used to check the correctness of the values. AR-SMT [24] expanded the idea of RMT on SMT processors. SRMT [21] improved the performance of AR-SMT where checkers were performed before stores. CRT [16] and CRTR [10] worked at the chip-level, but CRTR implemented recovery for CRT. These approaches need extra hardware support to validate the comparisons between the leading thread and the trailing thread.

Software level. The most attractive feature of software based error detection approaches is the portability to most systems without modifying the underlying hardware. For example, our proposed technique is applicable to most of the modern processors integrated with SIMD units. However, software approaches often come with performance and energy overhead as replication and re-execution are required. Compilers are often employed to automate error detection by intelligently inserting duplicates and checkers [11, 18, 23]. Khudia et al. [11] proposed error detection for soft computing applications using different protection schemes, e.g., traditional scalar instruction duplication, value checks, and no protection, for different computations.

A few work relied on using symptoms such as fatal traps, cache miss, branch misprediction, and application aborts to detect errors [8, 12, 33]. Shoestring [8] performed compiler analysis to identify the more vulnerable instructions (aka high-value in their paper) and protected them with instruction duplication. [12] improved Shoestring by using profiling information and they took into account anomalous microarchitectural behaviors for error detection. Symptom-based error detection techniques impose very little performance overhead compared to the full duplication techniques, but it scarifies error coverage. As the number of symptoms increases, the performance overhead also increases but the fault coverage starts saturating [11]. Compared to symptom-based techniques, we consider full instruction duplication for almost full error coverage.

Among all these solutions, EDDI and SWIFT are most closely-related to our proposed solution in the context of full instruction duplication. Our technique differs from them in the following aspects.

- Both EDDI and SWIFT were developed for ILP-friendly processors, e.g., SWIFT was running on Intel Itanium processor, to take advantage of ILP provided by interleaved instructions. Our solution targets most modern processors by exploiting the idle vector resources for redundancy.

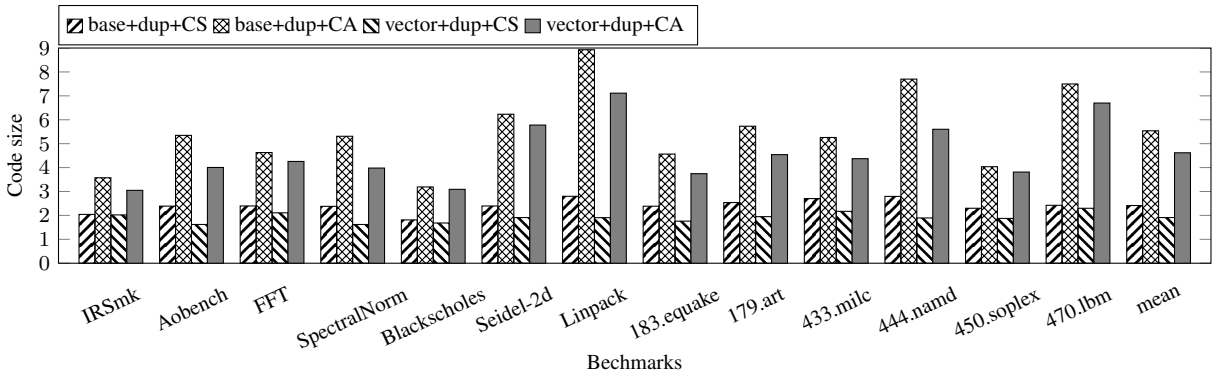


Figure 10: Binary size of scalar and vector instruction duplication with selective and full error checking.

- EDDI duplicated all instructions including stores. SWIFT avoided duplicating stores, but still needed to duplicate loads. Our solution eliminates duplication of memory accesses. Also, this work duplicates most library calls since there are SIMD intrinsics available for them.
- Our solution incurs much lower performance and energy overheads. It also reduces the code size significantly. For example, in order to achieve high error coverage, EDDI and SWIFT introduced more than 2.8x and 2.4x instructions, respectively. Our approach only leads to 1.9 times more instructions.

Hybrid approaches. While most of the aforementioned schemes focus on either hardware or software, some papers proposed to use hybrid approaches. CRAFT [22] provided reliability using a software-hardware hybrid approach where duplicated instructions and error checking codes were inserted with software methods and additional hardware (similar to RMT) was used to achieve higher error coverage. CRAFT provided better reliability with less performance loss than most software-only techniques but requiring hardware modification. Argus [14] is a hybrid approach to protect four types of invariants, namely control flow, data flow, computation, and memory. Hardware units were used to generate invariants online and perform comparisons between the invariants generated at runtime against the information collected from the compiler to detect faults. mSWAT [25] is also a symptom-based solution with little hardware support to detect anomalous software behaviors. However, compared to these techniques, our SIMD-based fault tolerance achieves almost full error coverage without the need of any special hardware support.

5. Conclusions and Future Work

This paper proposed a compiler framework to detect soft errors through instruction duplication using SIMD features of modern processors. It was implemented in the LLVM compiler. Performance, energy, and code size of compiled benchmarks were measured and the results show that, if error checking were only inserted at synchronization points, the SIMD assisted soft error detection only led to 1.21x performance slowdown and 1.14x energy consumption increase. This is significantly lower compared to prior scalar instruction duplication techniques. It also reduced the binary size by upto 20%.

Future work will focus on multicore processors and workloads written in other languages that are compatible with LLVM.

References

[1] Simatic s7-400. http://w3.siemens.com/mcms/programmable-logic-controller/en/simatic-s7-controller/s7-400/Pages/Default.aspx.

[2] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*, pages 196–207, 1999.

[3] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop reg; advanced architecture. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, 2005.

[4] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 333–345, 2006.

[5] A. Chaudhari, J. Abraham, and J. Park. A framework for low overhead hardware based runtime control flow error detection and recovery. In *Proceedings of the 2013 IEEE 31st VLSI Test Symposium (VTS)*, pages 1–6, 2013.

[6] Z. Chen, R. Inagaki, A. Nicolau, and A. V. Veidenbaum. Software fault tolerance for fpus via vectorization. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS XV)*, 2015.

[7] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *2011 IEEE International Reliability Physics Symposium (IRPS)*, pages 5B.4.1–5B.4.7, 2011.

[8] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 385–396, 2010.

[9] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 581–588, 2003.

[10] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 98–109, 2003.

[11] D. S. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2014.

[12] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 99–108, 2012.

[13] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of*

the International Symposium on Code Generation and Optimization (CGO), pages 75–86, 2004.

[14] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 210–222, 2007.

[15] K. Mohanram and N. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *International Test Conference (ITC)*, pages 893–901, 2003.

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–110, 2002.

[17] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 243–247, 2005.

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

[19] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, 2002.

[20] R. R. Rao, D. Blaauw, and D. Sylvester. Soft error reduction in combinational logic using gate resizing and flipflop selection. In *Proceedings of the International Conference on Computer-aided Design (ICCAD)*, pages 502–509, 2006.

[21] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2000.

[22] G. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August. Design and evaluation of hybrid fault-detection systems. In *Proceedings. 32nd International Symposium on Computer Architecture (ISCA)*, pages 148–159, 2005.

[23] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 243–254, 2005.

[24] E. Rotenberg. Ar-smt: a microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault-Tolerant Computing*, pages 84–91, 1999.

[25] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 122–132, 2009.

[26] N. R. Saxena and W. K. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Trans. Comput.*, 39(4):554–559, 1990.

[27] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*, pages 389–398, 2002.

[28] C. Slayman. Soft error trends and mitigation techniques in memory devices. In *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings - Annual*, pages 1–5, 2011.

[29] M. Snir, R. W. Wisniewski, J. A. Abraham, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing*, 28(2):129–173, 2014.

[30] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 207–216, 2010.

[31] R. Vemu and J. Abraham. Ceda: Control-flow error detection using assertions. *IEEE Transactions on Computers*, 60(9):1233–1245, 2011.

[32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 87–98, 2002.

[33] N. Wang and S. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, pages 30–39, 2005.

[34] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 264–275, 2004.