Impact of Loop Transformations on Software Reliability

Jason Cong and Cody Hao Yu
Computer Science Department, University of California, Los Angeles, CA, USA
{cong, hyu}@cs.ucla.edu

***Abstract*—Application-level correctness is a useful and widely accepted concept for many kinds of applications in this modern world. The results of some applications, such as multimedia, may be incorrect due to transient hardware faults or soft-errors, but they are still acceptable from a user's perspective. Thus, it is worthwhile to develop approaches to guarantee application-level correctness in software, instead of hardware, to reduce cost and save energy. Many previous research efforts presented solutions to identify parts of programs that may potentially cause unacceptable results, and placed error detectors to improve reliability. On the other hand, we observe that loop transformations have the ability to improve reliability. By applying suitable loop transformations, some critical instructions may become non-critical. In this paper we propose a metric to analyze the reliability impact of each loop transformation. Thus, we can guide a compiler to optimize programs not only for reliability improvement, but for energy saving. The experimental results show that our analysis perfectly matches the results of fault injection, and achieves a 39.72% energy saving while improving performance by 52.16% when compared with [1]. To our knowledge, this is the first work that considers a software reliability by loop transformations.**

I. Introduction

Transient hardware fault-induced soft-errors cannot be ignored due to today's technology scaling. Some instructions have to be re-executed once soft-errors are encountered. Since re-execution not only takes extra time, but extra energy, it results in soaring costs. Software reliability, i.e., the probability that software can be executed in a period of time and environment without failure [2], has became a critical problem in recent years.

Researchers in the hardware field attempt to solve this problem by developing novel circuits and architectures to protect against soft-error [3] [4] [5]. The goal of their work is to achieve numerical correctness of all program outputs, meaning that every bit of every output must be correct. This kind of correctness is also referred to as architectural-level correctness. Adopting the mechanism in hardware to assure architectural-level correctness implies additional hardware and energy overhead. On the other hand, for some fields of applications, such as multimedia and machine learning, even if the result is not perfectly correct, it is still acceptable from a user's perspective. This kind of correctness is referred as application-level correctness [6]. We shall discuss more recent work about application-level correctness in Section V.

To realize how an application can achieve application-level correctness, analyzing loop structure is inevitable, as most computations and memory accesses of a program involve loops. As a result, most compilers apply several kinds of loop transformations for improving performance. However, different loop structures have different impacts on reliability. Consequently, loop transformations may deteriorate software reliability if the impact of loop structure on reliability is not taken into account while optimizing the performance.

We use an example to illustrate the impact on reliability by different loop structures. Figure. 1(a) shows $ycc2rgb$, an important kernel in the JPEG decoder, in C language. This program takes an uncompressed color YCbCr image as input, and outputs an image with RGB colorspace. Consider a transient fault occurring at the first iteration of the inner loop in Figure 1(a). It causes one incorrect row in each primary color of the output image. In this case, we assume that it is unacceptable. On the other hand, Figure 1(b) presents the same program, but applies loop fission transformation. The fault still occurs at the first iteration of the inner loop. However, it results in

```c
void ycc2rgb(char **out, int char **in,
  int height, int width) {
  int M = height;
  int N = width;
  char **r = out;
  char **g = r + M*N;
  char **b = g + M*N;
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
      r[i][j] = ycc2rgb_red(in,i,j);
      g[i][j] = ycc2rgb_green(in,i,j);
      b[i][j] = ycc2rgb_blue(in,i,j);
    }
  }
  return ;
}
```

(a) The Original C Code

```c
void ycc2rgb(char **out, int char **in,
  int height, int width) {
  int M = height;
  int N = width;
  char **r = out;
  char **g = r + M*N;
  char **b = g + M*N;
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++)
      r[i][j] = ycc2rgb_red(in,i,j);
  }
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++)
      g[i][j] = ycc2rgb_green(in,i,j);
  }
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++)
      b[i][j] = ycc2rgb_blue(in,i,j);
  }
  return ;
}
```

(b) The C Code with Loop Fission

Fig. 1: ycc2rgb in C

only one incorrect row in primary color r, and this might become acceptable by considering the correctness threshold of fidelity metric such as PSNR (Peak Signal-to-Noise Ratio). As a result, although we introduce more loops to the program in this example, we decrease the possibility of rollback by reducing the number of instructions affected by the fault, so the reliability is able to be improved. In this paper, we present an evaluation metric to analyze the impact on software reliability due to loop transformations, and further apply suitable loop transformations for programs to improve program reliability and save energy. Our overall contributions list follows:

An evaluation metric for loop reliability. We present a metric by classifying each kind of loop-related instruction to evaluate the reliability of loops against soft-errors. We show that our classification, which is based on canonical natural loop structure, identifies the same set of critical instructions by previous work [7], but more efficiently.

Impact analysis on loop transformation. We analyze the reliability impact of each loop transformation by using the metric proposed in this paper, further deriving properties for improving reliability. In addition, we implemented a fault injector that has ability to evaluate our analysis. The experimental results also match the analysis we make in this paper.

Energy saving and performance improvement. Our experimental results show that by taking the suitable loop transformations according to our evaluation metric, the rollback frequency of a program can be reduced when encountering soft-errors. When compared with [1] we can achieve a 39.72% energy saving while improving performance 52.16% on average. We can also reduce 7.93% energy and improve 17.58% performance compared with the previous work [7].

II. Preliminaries

A. Fault Model

The fault we consider in this paper is single soft-error that results in transient single bit-flip in registers. It is usually generated by a cosmos ray, high-energy strikes, or random noise on circuits. For the fault in registers, one of the bits in the register may be flipped at any time during software execution and cause silent data corruption (SDC), or even crash programs. To make a conservative analysis, we assume that any fault in an instruction causes the worst outcome. In other words, the assumption is that a faulty instruction will affect the maximum number of output elements that it can, or may induce a program crash if the program and the platform that it runs on has no memory protection.

We want to mention that memories and caches are usually protected by ECC or parity in modern designs. Moreover, the fault that occurs at instruction registers, which affects program control flow, is able to be protected by control flow checking [8] [9]. Thus, the fault that we consider in this paper happens in the registers that store instruction inputs and outputs.

B. Critical Instruction

The output of a program is identified as an "elastic output" if the correctness value of this output that is defined by users is not unique [7]. Unlike the numerically corrected results required by scientific applications, the results generated by multimedia applications, such as images, videos, audios, and so on, do not need to be evaluated by matching each element precisely with the golden results at every pixel. Instead, results can only be interpreted qualitatively by a fidelity metric such as PSNR. In general, given an evaluated metric M, and an input/output pair of a program (I, O), the output O is identified as application-level correctness if $M(I, O) \leq T$, where T is a user-specified threshold.

In [7], the authors present a concept "*AFFECTER*" to identify critical instructions. Assuming that an elastic output has multiple elements, then an instruction is called N–*AFFECTER* of the output if an error occurring at any instance of the instruction affects at most N elements of the output. Given an application specific threshold T, an instruction is defined as a *critical instruction* if $N > T$; otherwise it is defined as a *non-critical instruction*. Please note that if the program or the platform that the program runs on has no illegal memory access protection mechanism, then the 1–*AFFECTER* instruction that performs the memory operation is identified as a critical instruction as well since it may cause a program to crash. To avoid this problem, [7] assumes the processor that the program runs on has memory access protection mechanism, as the case with processors such as SPARC v9 [10].

We use an example to demonstrate critical instructions. Figure. 2 shows the control flow graph (CFG) of the program presented in Figure. 1 (a). Blocks in CFG represent basic blocks, while edges present the control relationship between each basic block. According to the program, the iteration number of the outer loop and inner loop are M, and N, respectively. The block "body" contains the instructions that store output images (we omit details due to page

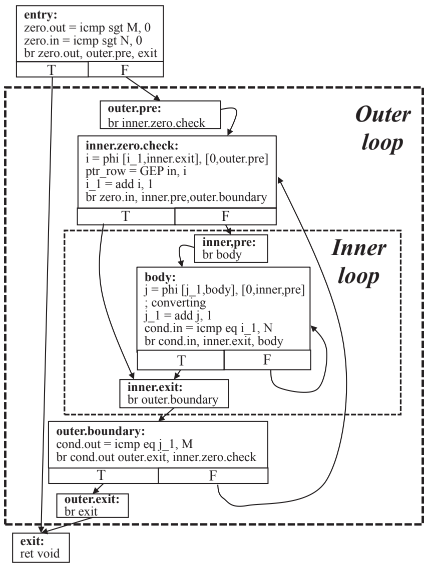

Fig. 2: Control Flow Graph of ycc2rgb

limit). As a result, instructions of the outer loop and inner loop are $3MN$–*AFFECTER* and $3N$–*AFFECTER*, respectively. We set the threshold as $3N$, meaning that the output image is allowed to have one row with incorrect R, G, and B color at most. Therefore, the instructions of the outer loop are critical instructions. On the other hand, since we assume that our platform has an illegal memory access protection, the instructions of the inner loop are non-critical instructions.

III. Reliability Analysis Under Loop Transformations

In this section we divide instructions into four categories based on their *AFFECTER* values [7]. Then we derive the form in terms of classified loop instructions to represent the critical instruction percentage of a program. Thus we can further analyze the reliability impact of each loop transformation by analyzing the mutated critical instruction percentage.

A. Loop Instruction Classification

The loops we discuss in this paper are canonical natural loops [11], which have only one entry basic block and one back-edge. Canonical natural loops consist of three parts: header, body, and exit. The header and exit parts are used for processing invariant data, such as iterator initialization and update. The loop body has one or more blocks, and is used for processing the computation, updating the iterator, and testing the boundary for branching. Then, we classify loop instructions based on their use.

Zero Iteration Instruction (ZI)**:** These kinds of instructions are adopted for testing if the number of iterations of a loop is zero or not. The loop is ignored in runtime to improve the performance if the number of iteration is zero. This part is eliminated by compilers if the iteration number of a loop is a constant value.

The ZI instruction for a loop that effects at most N elements of outputs is a critical instruction. The reason is that once a ZI instruction encounters a fault, the whole loop will be skipped in runtime. In addition, for a ZI instruction in the inner loop of a nested loop, if the iteration number of the inner loop is determined and will not be changed, compilers usually place the ZI instruction outside of the nested loop. For this case, the ZI instruction is always critical since once it has a fault, there is no chance to re-execute it again in the rest of iterations. On the other hand, if the iteration number of the inner loop depends on the outer loop, it has to be determined each

```
1        entry:
2(ZI)      zero.out = icmp sgt M, 0
3(ZI)      zero.in = icmp sgt N, 0
4(ZI)      br zero.out, outer.pre, exit
5        outer.pre:
6(J)       br inner.zero.check
7        inner.zero.check:
8(LI)      i = phi [i_1,inner.exit], [0,outer.pre]
9(P)       ptr_row = GEP in, i
10(LI)     i_1 = add i, 1
11(ZI)     br zero.in, inner.pre, outer.boundary
12       inner.pre:
13(J)      br body
14       body:
15(LI)     j = phi [j_1,body], [0,inner,pre]
16(Po)      ; converting
17(LI)     j_1 = add j, 1
18(LI)     cond.in = icmp eq i_1, N
19(LI)     br cond.in, inner.exit, body
20       inner.exit:
21(J)      br outer.boundary
22       outer.boundary:
23(LI)     cond.out = icmp eq j_1, M
24(LI)     br cond.out outer.exit, inner.zero.check
25       outer.exit:
26(J)      br exit
27       exit:
28         ret void
```

Fig. 3: ycc2rgb in LLVM IR

time before the iteration, then the ZI instruction must be placed right before the inner loop. Therefore, the ZI instruction is N–*AFFECTER* where N is the maximum iteration number of the inner loop.

Loop Iteration Instruction (LI): Loop iteration instructions are the instructions that are related to the control flow of the program. They include iterator initialization, update, and condition. For multimedia-related applications, an iterator of the loop that contains outputs is usually also used for indexing elements of outputs. We use LI^m to represent LI instructions that related to memory access. As a result, LI^m instructions are always critical if the system which the program runs on has no illegal memory protection. On the other hand, with a memory-accessing safe mechanism, LI instructions are the same *AFFECTER* of the loop that they belong to.

Jump (J): For every single canonical nature loop, there are always two jump instructions. One jumps from loop header into loop body; another jumps outside of the loop from the loop exit block. As mentioned in Section II-A, we assume that all branch targets are protected by control flow checking technology. Thus, jump instructions is always non-critical under this assumption.

Processing (P): The rest of instructions that relate to the outputs are identified as processing instructions. An instruction is defined as processing instruction P_i of loop i if it 1) does not belong to nigher ZI nor LI instructions, 2) has data dependency to loop i, and 3) has the same N–*AFFECTER* as loop i. Based on the definition, the processing instruction P_i is part of the loop invariant code of loop i. In addition, for the processing instruction in the innermost loop, which has dependency with the output, we define it as P_o instruction since it is always 1–*AFFECTER*.

A part of P_i that is related to memory access may cause a program crash if the system that the program runs on has no illegal memory-accessing protection. As a result, we further classify the process instructions of loop i as memory-related processing (P_i^m) and data processing (P_i^d) instructions. The P_i^m instruction is identified as a critical instruction with an unprotected system.

We use Figure 3 to demonstrate loop instruction classification. Figure 3 shows $ycc2rgb$ in LLVM intermediate representation (IR) for LLVM compiler infrastructure [11]. The beginning of each line indicates the type of instruction. Since the instruction at line 9 is a N–*AFFECTER* processing instruction and has data dependency with the inner loop, it is identified as a P_{inner}^d instruction. In addition, for the processing instructions at line 16, we identify these instructions as P_o instructions since they are 1–*AFFECTER*.

B. Critical Loop Identification

We define *critical loop* as a N–*AFFECTER* loop where N is larger than the user-defined threshold T; otherwise it is defined as *non-critical loop*. To analyze the reliability of loops in a program, we want to identify the relationship between loops and program outputs, which are arrays for multimedia applications. Algorithm 1 shows our methodology. We identify loop *AFFECTER* based on the following two cases.

First, if a loop writes the value to program outputs, then the *AFFECTER* of this loop is the product of the iteration numbers of its inner loops and itself, because the loop might effect at most N program output elements by N iterations. Therefore, we first generate a set of loops, $LoopwOutputSet$, to collect the loops that write the value to program outputs (line 2). Then we traverse these loops to compute the *AFFECTER* (line 3-9). Second, for the loop that does not write the value to program outputs, if the value produced by this loop has dependency with the program outputs, the *AFFECTER* of this loop and its inner loops are the same as the following loop with the same depth that writes the value to program outputs. For example, if the value produced by the loop L_A has dependency with the loop L_B, which writes to the program output, then this value is an invariant value for loop L_B. Thus, if loop L_B is a N–*AFFECTER* loop, then loop L_A is a N–*AFFECTER* loop as well. To deal with this case, we record the *AFFECTER* of each depth (line 8), and set the loops that do not write the value to outputs as the *AFFECTER* based on their depth (line 10-15). Finally, we identify critical loops by their *AFFECTER* (line 16-20).

Algorithm 1 Critical Loop Identification

```
Input: A set of loops and output instructions of a program.
Output: A set of critical loops.
 1: $Set\ DepthIterNum\ as\ 0$
 2: $LoopwOutputSet \leftarrow getLoopSet(OutputInst)$
 3: for each $Loop \in LoopwOutputSet$ do
 4:     $Loop.Aftr \leftarrow Loop.IterNum$
 5:     for each $InnerLoop \in Loop$ do
 6:         $Loop.Aftr \leftarrow Loop.Aftr \times InnerLoop.IterNum$
 7:     end for
 8:     $DepthIterNum[Loop.Depth] \leftarrow Loop.Aftr$
 9: end for
10: for each $Loop \notin LoopwOutputSet$ and $hasDep(OutputInst)$
    do
11:     $Loop.Aftr \leftarrow DepthIterNum[Loop.Depth]$
12:     for each $InnerLoop \in Loop$ do
13:         $InnerLoop.Aftr \leftarrow DepthIterNum[Loop.Depth]$
14:     end for
15: end for
16: for each $Loop$ do
17:     if $Loop.Aftr > Threshold$ then
18:         $Loop.Crit \leftarrow$ true
19:     end if
20: end for
```

C. Reliability Analysis Under Loop Transformations

In this section we analyze the reliability of each loop transformation. The *reliability* we mention in this paper is defined to be the percentage of the critical instructions of the total instructions which have dependency with the outputs. Specifically, the reliability for program $R(\mathcal{P})$ is defined as:

$$R(\mathcal{P}) = 1 - CP(\mathcal{P}) = 1 - \frac{CI_{\mathcal{P}}}{TI_{\mathcal{P}}} \tag{1}$$

where $CP(\mathcal{P})$ is the critical instruction percentage of program $\mathcal{P}$, $CI_{\mathcal{P}}$ is the number of critical instructions, and $TI_{\mathcal{P}}$ is the number of total instructions that have dependency with the program outputs. We use "total instruction" in the rest of this paper to be concise. For the instruction that has no dependency with program outputs, we do not consider them in this paper since they are able to be eliminated by compiler technologies such as code pruning.

According to Eq. 1 and Section III-A, for a program which runs on a system with illegal memory-accessing protection, LI_j, ZI_j, and P_j^d instructions for critical loop j are critical because they are *N–AFFECTER* where N is the iteration number of loop j, which is larger than the threshold defined by a user. Therefore, we are able to derive the critical instruction percentage as follows:

$$CP_w(\mathcal{P}) = \frac{(\sum_{i=0}^{CL} ZI_i + LI_i + P_i) + (\sum_{j=0}^{SNL} ZI_j)}{(\sum_{i=0}^{TL} J_i + ZI_i + LI_i + P_i) + P_o} \tag{2}$$

where CL is the total critical loop number in the program, and SNL is the non-critical loop with invariant (static) iteration number in runtime.

Prior work [7] proposed an algorithm, CIAP (Critical Instruction Analysis and Protection), to compute the *AFFECTER* of each instruction, and further identify whether the instruction is critical or not. However, the time complexity of the algorithm that [7] developed is O(N^3) where N is the number of basic blocks. Meanwhile, our algorithm needs only to identify critical loops, so the time complexity is O(N^2) where N is the number of loops that have dependency with the output. Consequently, our algorithm is more efficient than CIAP, while guaranteeing the equivalent critical instruction set as CIAP algorithm.

Theorem 1: Our proposed methodology has the same critical instruction set as the CIAP algorithm.

Proof: The CIAP algorithm identifies an instruction as a critical instruction based on the following rules:

1) The instruction is *N–AFFECTER* of the output if $N > T$, where N is the number of effected instructions by the instruction, and T is the user-defined threshold.
2) The instruction has dependency with the exit branch of the loop that contains at least one critical instruction.

If an instruction I is identified as a critical instruction by rule (1) of the CIAP algorithm, then $\prod_{k=1}^{n} IterNum(DL(k)) > T$, where DL is a set of loops that contains the output but not I. In this case, I will be classified as one of three instructions:

1) P_i of loop i, the outermost critical loop of DL.
2) ZI_i of loop i.
3) ZI_j of loop j, non-critical loop j with static iteration number within nested loop i.

Since all of these instructions are critical instructions, I is identified as critical by our method as well, and vice versa.

If an instruction I is identified as a critical instruction by rule (2) of CIAP algorithm, then it has dependency with the exit branch of the loop. As a result, I will be classified as an LI_i instruction of loop i by our method. Furthermore, since I is critical as identified by CIAP, loop i must have at least one critical instruction. It implies that loop i is a critical loop, so LI_i instructions of loop i are identified as critical instructions by our method as well, and vice versa. ■

In addition, we define *the different critical instruction percentage* after applying a loop transformation as:

$$CP'(\mathcal{P}) = \frac{\Delta CI_{\mathcal{P}}}{\Delta TI_{\mathcal{P}}} \tag{3}$$

Then, we have the following theorem.

Lemma 1: The critical instruction percentage is reduced after applying a loop transformation if and only if:

$$\frac{\Delta CI_{\mathcal{P}}}{\Delta TI_{\mathcal{P}}} < \frac{CI_{\mathcal{P}}}{TI_{\mathcal{P}}} \tag{4}$$

Proof: The condition of reducing the critical instruction percentage after applying a loop transformation is:

$$\frac{\Delta CI_{\mathcal{P}} + CI_{\mathcal{P}}}{\Delta TI_{\mathcal{P}} + TI_{\mathcal{P}}} < \frac{CI_{\mathcal{P}}}{TI_{\mathcal{P}}} \tag{5}$$

which is equivalent to the following equation:

$$\frac{\Delta CI_{\mathcal{P}}}{\Delta TI_{\mathcal{P}}} < \frac{CI_{\mathcal{P}}}{TI_{\mathcal{P}}} \tag{6}$$

■

We named Eq. 4 the "general form" since it can be used for any loop transformations to evaluate the reliability impact without loss of generality.

Subsequently, we analyze the reliability of each loop transformation under the system with an illegal memory-access protection mechanism based on Eq. 2. The reliability without memory protection mechanism is similar in concept, so we do not describe it in this paper. We will show that some loop transformations have recognized properties, but some others do not. For the transformations that cannot be concluded by their properties, we study the mutated critical instructions and apply the general form to analyze them. In addition, although we just describe some of the most widely used loop transformations due to page limit, our analyzed methodology can be easily extended for all other transformations.

1) Loop Tiling: Loop tiling, or loop blocking transformation, is a widely used loop transformation for data locality improvement. The order of data access can be rearranged to fit the size of a cache line by adding inner loops. Specifically, we add $K \times (LI + J + ZI)$ instructions where K is the number of inner loops that we add. Because the block size that we choose for loop tiling transformation is usually not very large for the size of a cache line, the added inner loops are usually non-critical loops.

The reliability impact of loop tiling transformation depends on the *AFFECTER* of inner loops after applying the transformation. Specifically, the *AFFECTER* of nested loops after loop tiling transformation is shown as follows:

$$AFFECTER_{tiling}(i) = B^{level(i)} \times AFFECTER(i) \tag{7}$$

where B is the size of the block we choose for tiling, and loop i is an inner loop of the nested loop; $level(i)$ means the level of loop i. We define the outermost loop as level 0, and the innermost loop as level $Lv - 1$ where Lv is the maximum loop nesting depth.

Theorem 2: Loop tiling guarantees reliability improvement if and only if no inner loop has changed from a non-critical loop to a critical loop.

Proof: The different instructions after applying loop tiling transformation can be represented as follows.

$$\begin{aligned} \Delta CI &= \sum_{j=k}^{Lv} (LI_j + P_j) \\ \Delta TI &= N \times (LI + J + ZI) \end{aligned} \tag{8}$$

where k is the outermost critical loop after applying loop tiling transformation. Therefore, the reliability can be improved if $k = Lv$, meaning that no inner loops changed from non-critical loops to critical loops. Otherwise, even if only one inner loop changed to critical, it may contain a lot of instructions that changed to critical as well. In this case, we have to use the general form to evaluate the reliability. ■

Take the case in in Figure 1 (a) as an example, loop i and loop j in Figure 1 (a) are $(3 \times M \times N)$ and $(3 \times N)$–*AFFECTER*, respectively. However, once we apply loop tiling transformation with block size $B \times B$, loop i is still $(3 \times M \times N)$–*AFFECTER* though loop j is changed to $(3 \times B \times N)$–*AFFECTER*. This may cause loop j to switch from non-critical to critical.

2) Loop Peeling: Loop peeling is a special case of loop splitting. It moves the first or the last iteration of a loop outside the loop so that some other loop transformations (such as loop fusion) are available to be applied. Since this transformation is usually applied for the innermost loop, it only adds the processing instructions that relate to data access (P^d) for the first or the last iteration of the loop, and any other loop instructions (LI) are not modified. Thus, loop peeling always improves reliability.

Theorem 3: Loop peeling always improves reliability.

Proof: The different instructions after applying loop peeling transformation can be represented as follows.

$$\begin{aligned} \Delta CI &= 0 \\ \Delta TI &= P_o \end{aligned} \tag{9}$$

Loop peeling for the innermost loop only adds data processing instructions P^d, and the innermost loop must be a non-critical loop or the application has no chance to achieve application-level correctness. As a result, the number of total static instructions has increased while keeping the number of critical instruction unchanged. In other words, loop peeling transformation reduces the critical instruction percentage and thus improves reliability. ■

Furthermore, from Theorem 3, a program without any loop is more reliable compared with the the same program but with loops, because it has no loop instructions so that all instructions are either 0–*AFFECTER* or 1–*AFFECTER*.

3) Loop Permutation (Interchange): Loop permutation is a loop transformation for a nested loop that changes the order of two or more iterators. Since it only changes the order of the inner loops of a nested loop, it does not change the instruction count of the program. Instead, it may change the *AFFECTER* of each loop. As a result, loop permutation can improve reliability when it changes one or more critical loops to non-critical loops.

Theorem 4: Loop permutation improves reliability if it reduces the critical loop number.

Proof: The number of different critical instructions after applying loop permutation can represented as follows:

$$\begin{aligned} \Delta CI =& \frac{\Delta CL}{|\Delta CL|}(\sum_{i=0}^{|\Delta CL|} ZI_i + LI_i + P_i) + \\ & \frac{\Delta SNL}{|\Delta SNL|}(\sum_{j=0}^{|\Delta SNL|} ZI_j) \\ \Delta TI =& 0 \end{aligned} \tag{10}$$

where ΔCL and ΔSNL represent the different number of critical loops and non-critical loops with a static iteration number. Since $\Delta TI = 0$, we can directly analyze the reliability by examining if the number of critical instruction has been reduced ($\Delta CI < 0$) or not.

$$-\frac{\Delta CL}{|\Delta CL|}\sum_{i=0}^{|\Delta CL|}(ZI_i + LI_i + P_i) > \frac{\Delta SNL}{|\Delta SNL|}\sum_{j=0}^{|\Delta SNL|}(ZI_j) \tag{11}$$

As we can see, Eq. 11 is satisfied if and only if $\Delta CL < 0$. Since $ZI + LI + P > ZI$, even if some loops with static trip count have been changed from critical (CL) to non-critical (SNL), the reliability can still be improved. ■

4) Loop Unrolling: Loop unrolling is a loop transformation technology used for optimizing program execution performance by reducing total iteration number while increasing the instruction count. As a result, loop unrolling transformation is a time-space trade-off and should be determined by either compiler or programmer. This transformation affects program reliability based on the *AFFECTER* of unrolled loop. We analyze the reliability of each situation as follows.

Unroll the innermost loop: The different instructions after unrolling the innermost loop is shown as follows.

$$\begin{aligned} \Delta CI &= 0 \\ \Delta TI &= N \times P_o \end{aligned} \tag{12}$$

where N is the unrolling times. According to Eq. 12, the only instructions that we increase when unrolling the innermost loop is processing instructions P_o, and P_o instructions are always non-critical since they are 1–*AFFECTER*. As a result, we can always improve program reliability by unrolling the innermost loop. On the other hand for the loop which is not the innermost loop, we separate it into two cases as follows.

Unroll a non-critical loop: If we attempt to unroll a non-critical loop within a nested loop, then we duplicate its inner loops which must also be non-critical loops. The different instructions of unrolling a non-critical loop is formed as follows.

$$\begin{aligned} \Delta CI &= 0 \\ \Delta TI &= \sum_{i=0}^{\Delta L}(LI_i + J_i + P_i) + N \times P_o \end{aligned} \tag{13}$$

For the system with memory protection, only ZI instructions might be critical instructions in a non-critical loop. However, we do not duplicate ZI instructions when unrolling loops since compilers usually let all duplicated loops share ZI instructions. As a result, we are able to improve reliability by unrolling non-critical loops.

Theorem 5: Unrolling non-critical loops or the innermost loop can always improve program reliability.

Unroll a critical loop: The reliability impact of unrolling a critical loop depends on the number of unrolled non-critical inner loops since we may duplicate zero or more critical and non-critical loops. The different instructions after unrolling a critical loop is shown as follows.

$$\begin{aligned} \Delta CI &= \sum_{i=0}^{\Delta CL}(LI_i + P_i) \\ \Delta TI &= \sum_{i=0}^{\Delta L}(LI_i + J_i + P_i) + N \times P_o \end{aligned} \tag{14}$$

where ΔCL is the different number of critical loops. Since unroll a critical loop cannot reduce the number of critical loops, ΔCL must be a positive value. We can see that unrolling a critical loop might duplicate its inner loops so that it increases every kind of instruction except ZI instructions, because they shared by all duplicated inner loops. As a result, this highly depends on the case of program and can only be evaluated by Eq. 4, the general form.

5) Loop Fission and Fusion: Loop fission transformation breaks a loop into multiple loops to split the original loop body into several parts. This transformation is primarily used for parallelizations since each independent loop can be allocated to a different thread to improve performance. On the other hand, the opposite of loop fission is loop fusion transformation. It merges multiple loops with same structure and index range into one loop. For the case that the body of each loop accesses the data within a dense range of memory, merging these bodies can improve data locality and further improve the performance.

The impact on reliability of loop fission also depends, even though each critical loop has the opportunity to be changed to a non-critical loop. The different critical instructions and total instructions after applying loop fission transformation are shown below:

$$\Delta CI = \sum_{i=0}^{\Delta CL} (LI_i + P_i) + \sum_{j=0}^{\Delta SNL} (ZI_j) \qquad \Delta TI = \sum_{i=0}^{\Delta TL} (LI_i + J_i + P_i) \tag{15}$$

where ΔCL is the number of changed critical loops, and ΔTL is the number of increased loops. It is no doubt that the reliability is improved if we eliminate all critical loops. However, since the instruction counts of each loop body vary, even if we significantly reduce the number of critical loops, we cannot guarantee that the reliability can be improved as long as there is at least one critical loop in the nested loop. Consequently, we cannot derive a strong relationship between loop fission transformation and program reliability. Instead, we have to evaluate the changed instructions to realize the reliability impact relying on the general form, Eq. 4, as well.

Theorem 6: Loop fission transformation is guaranteed to improve the reliability only if the loop that applied the transformation no longer has any critical loops.

Similarity, loop fusion transformation is the opposite operation of loop fission. The corresponding ΔCI and ΔTI are shown below.

$$\Delta CI = \sum_{i=0}^{\Delta CL} (LI_i + P_i) \qquad \Delta TI = \sum_{i=0}^{\Delta TL} (LI_i + J_i + P_i) \tag{16}$$

Since it always reduces instruction count, both ΔCL and ΔTI are less or equal to zero. As a result, the reliability is improved once it eliminates one or more critical loops.

Theorem 7: Loop fusion transformation is able to improve the reliability if it eliminates one or more critical loops.

D. Summary

In Table I we summarize the concepts that we analyzed in this section. For most loop transformations introduced in this section, we infer the necessary constraints to improve program reliability. Even for other transformations that have no clear properties of reliability impact, such as unrolling critical loops or splitting loop bodies, we still have the ability to evaluate them by using the general form.

TABLE I: Analysis Summary

Transformation	Guaranteed to improve reliability
Tiling	No inner loop changed from non-critical loop to critical loop.
Peeling	Always improves.
Permutation	Reduces critical loop number.
Unrolling	Unrolls the innermost or non-critical loops.
Fission	Eliminates all critical loops for applied nested loop.
Fusion	Reduces critical loop number.

IV. Evaluations

The applications that we adopt for evaluation are represented as LLVM intermediate representations (IR) which are provided by the LLVM compiler infrastructure [11]. The LLVM IR is a static single assignment representation that essentially models a RISC processor with infinite registers. This form is widely adopted since it makes use-def chains explicit so that developers are able to analyze and optimize programs easily. We assume that our applications are executed on commercial off-the-shelf processor platforms with ECC memory protection. We use the Multi2Sim simulator system [12] with 64KB L1, 4MB L2 caches, to evaluate performance, and adopt McPAT [13] for measuring energy overhead.

A. Fault Injection

To evaluate the critical instruction percentage of a program, we have to evaluate the *AFFECTER* of each instruction and identify if the instruction is critical or not. However, if we adopt random fault injection to inject a fault into an instruction, the fault might be masked by the following instructions and cannot be observed. For example, the instruction $R_a = R_b \& 0x1$ ignores the value of register R_b except for the least significant bit. As a result, even if one bit of R_b has been flipped due to a soft-error, the value of R_a is still correct if the error bit of R_b is not the least significant bit. We define the instruction that has ability to mask faults as a *maskable instruction*. To avoid this problem, we can keep injecting faults into a certain instruction until the fault has been observed through the output. However, it may take too much time to evaluate all instructions, and we cannot differentiate between a non-observable fault and a 0–*AFFECTER*. As a result, we identify the instructions set, which the faults occurring in instructions within the set may be masked by the specific instruction. Then we inject an observable fault into instructions that will not be masked by others. By the fault injector, we are able to evaluate the *AFFECTER* of all instructions.

TABLE II: Maskable Instructions

Categories	Sample instructions
Logical	and, or, xor
Shift	shl, lshr
Conversion*	trunc, fptosi
Condition	icmp, fcmp

* Doesn't include bit-extension instructions.

We summarize maskable instructions in Table II. Note that some conversion instructions, such as bit truncation, have the ability to mask faults; however, others, such as bit-extension, cannot. The reason is that this kind of instruction reserves all bits from input, so it does not eliminate the error from the input register.

The algorithm of the fault injection is presented in Algorithm 2. The fault injector first initializes a set, $ToInjectSet$, to collect instructions that will be injected a fault (line 1). Then it generates a golden result to be compared later (line 2). Subsequently, we traverse the dependency graph of each instruction I from the program output to see if instruction I reaches a maskable instruction MI based on Table II before reaching the program output or not (line 6). If yes, then the fault injector will not inject a fault into instruction I since it will likely be masked by maskable instruction MI. Instead, the fault injector puts instruction I to the group of maskable instruction MI (line 8). A group is a set of instructions whose faults might be masked by the same instruction. On the other hand, if instruction I does not reach any maskable instruction, then it must have capability to propagate a fault to the program output. As a result, fault injector puts instruction I to $ToInjectSet$ (line 10) and creates a group of instruction I (line 11). We want to mention that if a maskable instruction MI_A reaches another maskable instruction MI_B, then since we traverse instructions from the program output, MI_A will be put into the group of MI_B. Thus, the rest instructions which reach maskable instruction MI_A will be put into the group MI_B as well.

After traversing instructions, the fault injector injects a fault to each instruction in $ToInjectSet$ and computes the incorrect result number by comparing the result with the golden one as the *AFFECTER*. Finally the fault injector assign the *AFFECTER*, $Aftr$, of all instructions in the group (line 15-19). By adopting the fault injector, we have the capability of knowing the accuracy *AFFECTER* of every instruction, and further, can evaluate the properties presented in Section. III-C.

Algorithm 2 Fault Injection and Simulation for Critical Analysis

```
Input: An instruction level program $\mathcal{P}$.
Output: The AFFECTER of each instruction.
 1: $ToInjectSet \leftarrow NULL$
 2: $GoldenResult \leftarrow Execute\ program\ without\ fault$
 3: $ToTraverseSet.push(Output(\mathcal{P}))$
 4: while $ToTraverseSet! = NULL$ do
 5:     $I = ToTraverseSet.pop$
 6:     $MI = reachMaskableInstruction(I)$
 7:     if $MI! = NULL$ then
 8:         $Group(MI).add(I)$
 9:     else
10:         $ToInjectSet.add(I)$
11:         $CreateGroup(I)$
12:     end if
13:     $ToTraverseSet.push(I.input)$
14: end while
15: for each $I \in ToInjectSet$ do
16:     $Inject\ a\ fault\ into\ I$
17:     $Result \leftarrow Execute\ program\ with\ a\ fault$
18:     $Group(I).Aftr \leftarrow Diff(Result, GoldenResult)$
19: end for
```

B. Experimental Results

1) Loop Impact Analysis: We use $ycc2rgb$ as the case in this experiment to evaluate the consistency of our analysis with the fault injector. We manually perform several loop transformations and evaluate the reliability impact based on the analysis that we presented in Section III. Table III shows the experimental results. Columns 2-3 in Table III show the number of total instructions and basic blocks, respectively. Column 4 provides the critical loop number, while column 5 represents the critical instruction count evaluated by the fault injector. Column 6 presents the critical instruction percentage of each transformation which is inversely proportional to the reliability.

TABLE III: Results of Fault Injection

Trans-formation	#Inst	#BB	#CL	#CI	CP
Baseline	176	13	2/2	56	31.82%
Unrolling*	433	20	5/5	86	19.86%
Tiling	244	19	2/4	70	28.69%
Fission	375	35	3/6	62	16.53%
Peeling	219	13	2/2	47	21.46%
Permutation	176	13	2/2	64	31.82%
Combination	905	56	8/13	182	20.11%

* Here we unroll outer loop (critical) by 4 times.

For loop unrolling and loop fission, according to general form, $\Delta CI/\Delta TI$ for both cases are smaller than the baseline so that the reliability is able to be improved. Loop tiling keeps the reliability of the inner loop unchanged, so reliability is improved by Theorem 2. Loop peeling always improves reliability based on Theorem 3. On the other hand, loop permutation does not reduce the number of critical loops, so we cannot benefit from Theorem 4. Therefore, the reliability remains the same. Finally, we perform a combinational case by applying tiling, peeling, and unrolling transformations in order. Since applying tiling and peeling to $ycc2rgb$ improve reliability, the combinational case has lower critical instruction percentage compared with the cases applied the transformations individually. However, we unroll the outermost loop, which is a critical loop, after applying these two transformations, and unroll a critical loop degrades reliability in this case. Consequently, depending on different program and applied loop transformations, the reliability might be improved or degraded. In addition, the order of applied transformations also influences the reliability.

2) Energy Efficiency Analysis: To measure how loop transformation affects energy efficiency, we inject faults into the program by random fault injection to simulate practical situations; i.e., faults may be masked by maskable instructions. As a result, for each instruction in a program, we insert a faulty instruction with a given soft-error probability to trigger it, so the fault is probably activated in a random instance of an instruction. To facilitate the experiment, we use a 0.01% soft-error rate. We adopt the error detector proposed by [1] which duplicates instructions and compare results before writing into memory. If the fault is detected, then the program rolls back to the beginning of the basic block and executes again.

We perform our experiment for multimedia applications from MiBench [14] and MediaBench II [15]. Ycc2rgb, IDCT, and Huffman are important kernels of the JPEG decoder. Susan is an image recognition program for detecting edges and corners in images. ADPCM, MPEG2, and H264 are decoders for either speech or videos. Rician is a program used for MRI denoising. All applications are compiled with $-O2$ optimization as baselines, and applied suitable loop transformations for observing the effect of energy and performance.

TABLE IV: Energy and Performance Comparison

Benchmark	CI Red.	Energy saving		Perf. improvement	
		[1]	[7]	[1]	[7]
Ycc2RGB	36.80%	51.45%	3.71%	70.93%	7.45%
IDCT	18.10%	37.47%	1.25%	54.36%	6.76%
Huffman	6.85%	49.87%	3.67%	44.69%	0.34%
ADPCM	78.06%	42.27%	22.78%	83.57%	50.98%
Rician	42.66%	64.62%	15.21%	54.30%	27.87%
Susan-edges	43.05%	22.39%	10.10%	53.97%	20.02%
Susan-corner	45.48%	55.90%	5.95%	41.32%	25.91%
MPEG2	59.61%	27.55%	6.95%	58.08%	21.39%
H264	53.06%	5.95%	1.76%	15.17%	-2.51%
Average	42.63%	39.72%	7.93%	52.16%	17.58%

Table IV shows energy saving and performance improvement while achieving application-level correctness with respect to [7] and [1]. For each application, we set the threshold as 10%. Column 2 shows the ratio of reducing critical instructions after loop transformations. Since we spread the impact of critical instructions efficiently, we are able to reduce 42.63% of critical instructions on average. Columns 3-4 and columns 5-6 provide energy saving and performance improvement ratios of each case compared with previous works, respectively. As can be seen, as long as we reduce the percentage of critical instructions, we also reduce the overhead of error detectors. Consequently, energy and performance are improved due to roll back frequency reduction. Here we mention that since [1] attempts to achieve numerical correctness, it treats all instructions as critical instructions. Therefore, error detector induced overhead causes relatively poor results for performance and energy.

For some cases that are mainly composed by non-critical loops, such as IDCT and Huffman, even if we apply several suitable loop transformations for reliability improvement, the benefit is limited. On the other hand, the kernel of ADPCM has no nested loop, so we can only apply loop unrolling and loop peeling. Despite that, since we unroll the “innermost” loop when we unroll a single-level loop, the reliability is advanced based on Theorem 5. Unlike ADPCM, Rician has many critical nested loops, implying that we have more opportunities to reduce the critical instructions percentage. Similarly, the kernels in Susan have a lot of affine access within critical nested loops, which heavily affects control flow. In this case, we mainly adopt loop tiling to reduce the critical instruction percentage. In addition, the kernel in MPEG2 performs some blocking computations by using loops with constant iteration numbers. For this case, we fully unroll these loops, which results in both energy saving and performance improvement. In contrast, H264 has many huge loops for processing the whole video streaming. Thus, even though our methodology reduces a considerable critical instruction percentage, the increased instructions cause non-negligible overhead and cannot be eliminated by reducing roll back frequency. Accordingly, the performance and energy cannot be improved well.

V. Related Works

Recent work has proposed solutions for analyzing soft-error problems to ensure application-level correctness by developing fault injectors. Fault injectors help researchers determine the critical instructions that either cause program crashes or generate unacceptable output to analyze program reliability. LLFI [16] and KULFI [17] are high-accuracy, open-source LLVM fault injectors used for randomly injecting faulty instructions into programs in LLVM IR. [18] designed a fine-grained soft-error fault injector, F-SEFI, which leverages the virtual machine and its hypervisor to reduce environment and input data dependency while injecting faults into programs. Relyzer [19] identified equivalent fault groups so that it can reduce the complexity of injecting faults. Based on the contributions of fault injectors, researchers are able to further improve the soft-error resilience of applications. [20] generalized some classes of faulty instructions that lead to error derating. [21] targeted improving program reliability by exploring fault masking. The author in [21] categorized instructions that have the ability to mask faults, and further presented an optimization scheme that can be integrated into compilers to improve program reliability by using the characteristics of fault masking and error statistics from a fault injector.

On the other hand, some research aims to realize egregious positions or critical instructions in a program by using fault injection or static program analysis, and inserts detectors/protectors into the program to reduce SDC (silent data corruption) in runtime. [6] set the checkpoint periodically to PC, register file, and program stack at the top of each loop manually. [22] analyzed the program-level properties of SDC and inserted a low-cost detector, assertion, into the program. [23] provided an analysis and transformation method to categorize computations and protect them by using either instruction duplication or value checking. [24] presented ranking and selection algorithms based on the result of fault injectors for placing detectors to cover a wide range of SDCs. [25] developed a compiler-time algorithm to instrument checksum code into applications for detecting memory errors. [7] presented a novel concept of critical instruction definition, and used a program dependency graph to identify critical instructions. Furthermore, critical instructions were protected by duplicating monitoring these instructions in runtime. The objective of all these research work mainly focus on identifying critical regions, inserting error detectors and recovery mechanisms to ensure application-level correctness. In contrast, in addition to identifying critical loops accurately, our methodology reduces the number of critical loops by loop transformations. Consequently, the cost of code instrumentation can be reduced as well.

VI. Conclusions and Future Work

In this paper, we present a metric to analyze soft-error induced reliability impact for loop transformation based on canonical natural loop. We first classify loop instructions relying on their use. According to the classification, we further derive critical instruction percentage to represent software reliability. Then we are able to analyze the impact of every loop transformations. Our analysis result shows that every loop transformation has the potential to improve reliability under some specific conditions. Equipped with such analysis capability, we evaluate available transformations for programs and apply suitable transformations to advance reliability. Our experimental results show that applying suitable transformations benefit not only reliability but also energy.

There are several extension researches that we can explore in the future. Since our methodology works well with single thread programs, and the problem with multi-threading becomes more complicated, we will extend our methodology to work adequately for parallel programs. In addition, we plan to adapt the methodology for broadening the target application domains so that it can be adopted for other important applications such as machine learning and artificial intelligent applications.

VII. Acknowledgment

This work is partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by NSF grants CCF-0903541, and CF-1524127. The authors would like to thanks Louis-Noel Pouchet for his valuable comments.

References

[1] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[2] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[3] N. Avirneni *et al.*, "Low overhead soft error mitigation techniques for high-performance and aggressive systems," in *DSN*, 2009, pp. 185–194.

[4] H.-M. Chou *et al.*, "Soft-error-tolerant design methodology for balancing performance, power, and reliability," *IEEE Transactions on Very Large Scale Integration Systems*, 2014.

[5] S. Mitra *et al.*, "Combinational logic soft error correction," in *ITC*, 2006, pp. 1–9.

[6] X. Li *et al.*, "Application-level correctness and its impact on fault tolerance," in *HPCA*, 2007, pp. 181–192.

[7] J. Cong *et al.*, "Assuring application-level correctness against soft errors," in *ICCAD*, 2011, pp. 150–157.

[8] D. S. Khudia *et al.*, "Low cost control flow protection using abstract control signatures," in *LCTES*, 2013, pp. 3–12.

[9] N. Oh *et al.*, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, pp. 111–122, 2002.

[10] "The sparc architecture manual, version 9," http://developers.sun.com/solaris/articles/sparcv9.pdf.

[11] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[12] R. Ubal *et al.*, "Multi2sim: A simulation framework for cpu-gpu computing," in *PACT*, 2012.

[13] S. Li *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009, pp. 469–480.

[14] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001, pp. 3–14.

[15] J. E. Fritts *et al.*, "Mediabench ii video: Expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301 – 318, 2009.

[16] A. Thomas *et al.*, "Llfi: An intermediate code level fault injector for soft computing applications," *SELSE*, 2013.

[17] V. Sharma *et al.*, "Towards formal approaches to system resilience," in *PRDC*, 2013, pp. 41–50.

[18] Q. Guan *et al.*, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *IEEE International on Parallel and Distributed Processing Symposium*, 2014, pp. 1245–1254.

[19] S. Sastry Hari *et al.*, "Relyzer: Application resiliency analyzer for transient faults," *MICRO*, vol. 33, no. 3, pp. 58–66, 2013.

[20] J. Cook *et al.*, "A characterization of instruction-level error derating and its implications for error detection," in *DSN*, 2008, pp. 482–491.

[21] M. Shafique *et al.*, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *DAC*, 2013, pp. 1–9.

[22] S. K. S. Hari *et al.*, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, 2012, pp. 1–12.

[23] D. Khudia *et al.*, "Harnessing soft computations for low-budget fault tolerance," in *MICRO*, 2014, pp. 319–330.

[24] A. Thomas *et al.*, "Error detector placement for soft computation," in *DSN*, 2013, pp. 1–12.

[25] S. Tavarageri *et al.*, "Compiler-assisted detection of transient memory errors," *SIGPLAN Not.*, vol. 49, no. 6, pp. 204–215, Jun.