Procedure Call Duplication: Minimization of Energy Consumption with Constrained Error Detection Latency

Nahmsuk Oh and Edward J. McCluskey
Center for Reliable Computing – Stanford University, Stanford, California
{nsoh, ejm}@crc.stanford.edu

Abstract

This paper presents a new software technique for detecting transient hardware errors. The objective is to guarantee data integrity in the presence of transient errors and to minimize energy consumption at the same time. Basically, we duplicate computations and compare their results to detect errors. There are three choices for duplicate computations: (1) duplicating every statement in the program and comparing their results, (2) re-executing procedures with duplicated procedure calls and comparing the results, (3) re-executing the whole program and comparing the final results. Our technique is the combination of (1) and (2): Given a program, our technique analyzes procedure call behavior of the program and determines which procedures should have duplicated statements (choice (1)) and which procedure calls should be duplicated (choice (2)) to minimize energy consumption while controlling error detection latency constraints. Then, our technique transforms the original program into the program that is able to detect errors with reduced energy consumption by re-executing the statements or procedures. In benchmark program simulation, we found that our technique saves over 25% of the required energy on average compared to previous techniques that do not take energy consumption into consideration.

1. Introduction

Transient faults can cause abnormal behavior in computer systems. Radiation, electromagnetic interference and power glitches are some of the causes of transient errors. For example, in radiation environments, alpha-particles, cosmic rays and solar wind flux can cause *single event upsets* (SEU), which can cause the state of a memory cell to change from 0 to 1 or from 1 to 0. Radiation effects in microprocessors and memory in a space satellite were observed in the Stanford ARGOS project [1]. In a radiation environment, high reliability and dependability can be obtained if the system includes fault tolerance such as error detection and recovery. However, fault tolerance requires redundancy: hardware or software redundancy. Redundant hardware or software may cause execution time overhead, energy consumption or area overhead. Power consumption or energy dissipation overhead is especially important in mobile computing applications and space computing applications. Mobile computing applications require low energy consumption for longer battery life. In space applications, only limited power is available to the system; thus, we need to use low power components to maximize the utilization of the available energy. In this paper, we present a new Software Implemented Hardware Fault Tolerance (SIHFT) technique that optimizes energy consumption of the program execution while controlling error detection latency.

1.1. Related work

When we use Commercial-Off-The-Shelf (COTS) components in the system and cannot modify an existing hardware design for fault tolerance, Software Implemented Hardware Fault

1063-6722/01 $17.00 © 2001 IEEE

Tolerant (SIHFT) techniques are especially important. Several SIHFT techniques targeting specific faults have been proposed. For example, a number of control flow checking techniques [2] have been developed using additional hardware such as watchdog processors; control flow checking by software signatures in [3] is a pure software technique using software signatures.

Researchers also have studied pure software methods for concurrent error detection (CED) that do not target specific faults. Basically, they repeat computations in such a way that many types of errors (memory errors, functional unit errors, and some control flow errors) can be detected by comparing computation results. The idea of duplicating instructions in a VLIW processor was previously investigated by Holm and Banerjee [4]. They duplicated ALU instructions to detect errors in data paths. In [5], a more optimized instruction duplication technique detecting memory errors was proposed. However, they did not take energy consumption into consideration.

1.2. Our approach and contribution

So far, there have been few studies combining fault tolerance and low power software techniques. The basic concept of our technique is to duplicate *procedure* execution. A *procedure* is a named sequence of statements executed as a unit. Our goal is to detect transient errors in the processor. Duplicated computation will detect transient errors in data path units such as a register file, an ALU, a pipeline execution unit, and floating point unit (FPU). Furthermore, because the correct code is preserved by ECC in cache and memory, a transient error in the instruction fetch unit and the decoding unit will be detected when a procedure is called and executed twice. For example, suppose a transient error corrupts the original execution of the procedure. Because of ECC, however, the correct code can be fetched and used during the duplicated execution of the procedure. Thus, we can obtain a correct result in the duplicated execution and detect the error by comparing the incorrect result from the original execution and the correct result from the duplicated execution.

Our technique is different from the re-computation techniques in [5] and [6], which duplicate computations at assembly level and compare the two results by comparison instructions. By selectively duplicating procedure calls instead of duplicating every instruction, our approach to minimizing energy dissipation is to reduce the number of clock cycles, cache access, and memory access. The number of additional clock cycles is reduced because we reduce the number of comparisons by checking the computation results after the original and duplicated procedure execution, instead of checking the results right after executing every duplicate instruction. The code size is reduced because we do not duplicate some of the code of procedures. Instead, we execute the code of those procedures twice by calling the procedures twice and comparing the computation results to detect errors. Moreover, reducing the number of comparisons will decrease the number of data accesses to the data cache and the memory resulting in reduced energy consumption. In addition, if the code size is reduced, we can lower the probability of an instruction cache miss and reduce energy dissipation for fetching instructions from the cache to the processor, or moving instructions from the memory to the cache.

However, there is a trade-off between energy saving and error detection latency. The shortest error detection latency can be achieved by the instruction level duplication because we can detect a computation error by executing the original and duplicated instructions and comparing the two results; thus, the error detection latency is less than three cycles assuming each instruction execution takes one cycle. However, in procedure call duplication, we postpone the comparison of the results until after executing the callee procedure twice; thus, the worst case error detection latency for this procedure execution is:

$$\begin{aligned}&\text{execution time of the original and duplicated procedures} + \text{comparison time}\\&= 2\times\text{total execution time of the procedure} + \text{comparison time}\end{aligned}.$$

In other words, if an error occurs during the computation, it cannot be detected until the procedure completes its duplicated execution and compares the result from the duplicated execution with the result from the first execution. However, longer error detection latency will reduce the number of comparisons inside the procedure and, therefore, will save energy.

In this paper, we discuss an optimization problem of minimizing energy consumption and reducing error detection latency. We describe a new algorithm of duplicating procedure calls for error detection, show how to minimize energy consumption, and find an optimal solution considering the energy consumption and the error detection latency as parameters.

2. Procedure Call Duplication

2.1. Representation of procedure calls

A call graph (CG) represents the calling behavior of a program (Fig. 2.1 (a)). A graph vertex represents a procedure of the program. A CG has an arrow $X \rightarrow Y$ iff procedure X has a procedure call statement that calls procedure. The start node of the CG is the main procedure of the program. A *dynamic call tree* (DCT) in Fig. 2.1 (b) is a tree in which each tree vertex represents a single procedure activation and each edge represents the relationship between the caller and callee procedures. It shows a run-time representation of the calling behavior of a program and precisely records invocation of the procedures. For example, the DCT in Fig. 2.1 (b) can show the sequence of procedure invocation: *ADEBCEBC.* However, the CG in Fig. 2.1 (a) cannot represent how many times *A* calls *B* during run time.

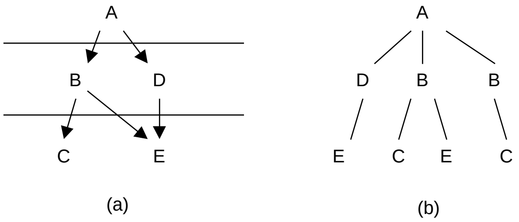
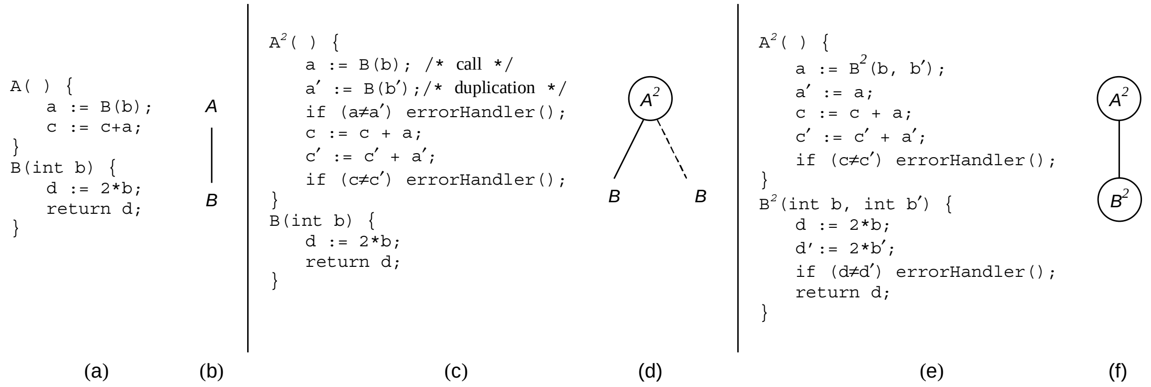


Fig. 2.1 (a) A call graph (CG) showing calling behavior of a program, (b) A dynamic call tree (DCT) showing run-time calling behavior of (a).

2.2. Duplication Basics

We can duplicate computations at different levels of program execution: instruction level, procedure level or program level. On one hand, the finest level of duplication is *instruction level duplication* in which an individual instruction is duplicated. On the other hand, the coarsest level of duplication is *program duplication* in which the whole program or program execution is duplicated. In *selective procedure call duplication*, some of the procedures have their statements duplicated, but others do not.

2.2.1. Procedure call statement duplication: Let us suppose that procedure *A* calls procedure *B* which returns a computation result after its completion. We illustrate the details in Fig. 2.2. A simple source code is shown in Fig. 2.2 (a) where *A* calls *B.* Its DCT is shown in Fig. 2.2 (b). We duplicate the statements in *A* and call B twice to compare the first and second results returned by *B*. Let us denote A^2 as a procedure that has duplicated statements for error detection and produces the same result as the original procedure *A*. The source code for A^2 is shown in Fig. 2.2 (c), in which *A* calls *B* with the variable `b` first, and then with the duplicated variable `b′`. The returned results are compared, and an error can be detected by mismatch of the results. Note that the code size of A^2 including comparison statements is more than twice the original code size of *A.* A DCT representing this calling behavior is shown in Fig. 2.2 (d). In the DCT, we indicate the second procedure call with a dotted edge. In addition, a procedure that has duplicated statements and comparison statements for error detection is denoted as a circled node. In Fig. 2.2 (d), all statements in A^2 are duplicated; thus, A^2 is circled. However, *B* does not have any duplicated or comparison statements for error detection; thus, *B* is not circled. If an error occurs during the computation in *B*, it can be detected after the second result from *B* is available.



```
A( ) {
    a := B(b);
    c := c+a;
}
B(int b) {
    d := 2*b;
    return d;
}
```

```
A^2( ) {
    a := B(b); /* call */
    a' := B(b');/* duplication */
    if (a≠a') errorHandler();
    c := c + a;
    c' := c' + a';
    if (c≠c') errorHandler();
}
B(int b) {
    d := 2*b;
    return d;
}
```

```
A^2( ) {
    a := B^2(b, b');
    a' := a;
    c := c + a;
    c' := c' + a';
    if (c≠c') errorHandler();
}
B^2(int b, int b') {
    d := 2*b;
    d':= 2*b';
    if (d≠d') errorHandler();
    return d;
}
```

Fig. 2.2. (a) Procedure *A* and *B* in original source code, (b) *A* calls B, (c) Duplicated statements in A^2 that calls *B* twice, (d) A^2 calls *B* twice and compares the results. The duplicated call is shown as a dotted edge, (e) Duplicated statements in both *A* and *B*, (f) Both A^2 and B^2 have duplicated statements, and A^2 calls B^2.

If we desire shorter error detection latency for errors that may occur in *B*, we have to duplicate the statements in *B*. The source code in which statements in *B* are also duplicated is shown in Fig. 2.2 (e). A DCT of this case is shown in Fig. 2.2 (f). Since A^2 and B^2 have comparison statements inside them, an error that occurs inside these procedures is immediately detected; thus, A^2 and B^2 are circled. However, shorter error detection latency is achieved by increasing the code size (increasing the number of statements) as we can observe that the code size in Fig. 2.2 (e) is greater than the code size in Fig. 2.2 (c).

3. Transformation Algorithm

Our optimization involves two objectives: reducing error detection latency and minimizing energy consumption. However, as we have seen in the previous section, there is a trade off between error detection latency and energy consumption. In this paper, we are particularly interested in the following optimization problem:

Minimize energy consumption under an error detection latency constraint.

For example, in critical application programs or real time programs, there is a requirement that we have to detect errors within a certain amount of time. In this case, the optimization problem is to reduce energy consumption under an error detection latency constraint.

We have developed an heuristic algorithm for this optimization problem and show it in Fig.3.1. In this algorithm, we denote the execution time of procedure *X* by λ_X and the error detection latency constraint by λ.

4. Simulation Experiment

We used the Wattch architectural-level power simulator [6] to evaluate our technique. Wattch estimates CPU power consumption based on a suite of parameterizable power models for different hardware structures. In the Wattch simulator, we implemented a procedure profiling so

```
ProcedureCallDuplication (program p, height m)
{
Build a CG of the program p.                    /* the first pass */
For each leaf node procedure X in p
    MarkIfLatencySatisfied(X)
d := 1;                                         /* the second pass: start node (main procedure): depth d = 1 */
while d ≤ the height of CG do
    For each procedure X : depth(X) = d
        If X is unmarked
                Duplicate statements
                If there is a procedure call calling procedure Y {
                    If Y is marked
                        Duplicate procedure call statement
                    else
                        Duplicate arguments in the procedure call statement
    d++
}
MarkIfLatencySatisfied(procedure X)
{
If $2\lambda_X < \lambda$
    Mark this procedure X
else
    For each parent node procedure Z
        MarkIfLatencySatisfied(Z)
}
```

Fig. 3.1. Selective Procedure Call Duplication algorithm.

that we can measure cycle and energy consumption for each procedure while we simulate the program execution. We configured our simulator based on the datasheet of the Intel StrongArm® microprocessor, one of the most popular embedded microprocessors, and conducted a simulation experiment.

We chose 5 benchmark programs from a public benchmark web site [7] to evaluate the effectiveness of our technique. We selected programs that have core routines used in embedded systems such as MPEG applications or communication systems. Those are MPEG – an audio MPEG compress decoder, FFT – a fast Fourier transform, Huff – a huffman decoder, Sort – an insertion sorting algorithm, and LZW – a lossless compression and expansion program.

We first build a CG for each benchmark program using *cg*, one of the SUIF compiler tools building a call graph of a program [8]. In Fig. 4.1(a), we show the CG of Sort program with execution time of each procedure. In Fig. 4.1(b), we show the transformed programs with different error detection latency constraints (λ = 5 *nsec*, 1 *msec*, 10 *msec*). If λ = 5 *nsec*, we cannot duplicate any procedure call because the shortest execution time of the procedure is 10 *nsec*. If λ = 1 *msec*, the procedure call to *C* can be duplicated. Similarly, if λ = 10 *msec*, the procedure call to *D* can be also duplicated. The CGs of the rest of the benchmark programs are also shown in Fig. 4.1.

With different values of error detection latency constraints, we applied the algorithm to benchmark programs to optimize energy consumption of the program execution. In Fig. 4.2 (a), we show the energy saving with different values of error detection latency constraints compared to the previous technique in [5]. The graph in Fig. 4.2 (a) shows the percentage of energy savings for the error detection latency from 100 *usec* to 10 *msec*. The X axis is in log scale because the execution time of the programs are different. For example, 1024 point FFT completes the whole computation in 1.63 *ms*. On the other hand, Huff takes more than 10 *ms* to complete its computation. The graph shows that energy consumption is reduced as we loosen the error detection latency constraint. Therefore, our algorithm will find an optimal point where energy consumption is minimized satisfying the error detection latency requirement. Finally, Fig. 4.2 (b) shows maximum power savings in which error detection latency is the longest possible value. The maximum is 56.3% in FFT and the minimum is 10.4% in Huff. The maximum power saving, on average, is 26.2%.

1063-6722/01 $17.00 © 2001 **IEEE**

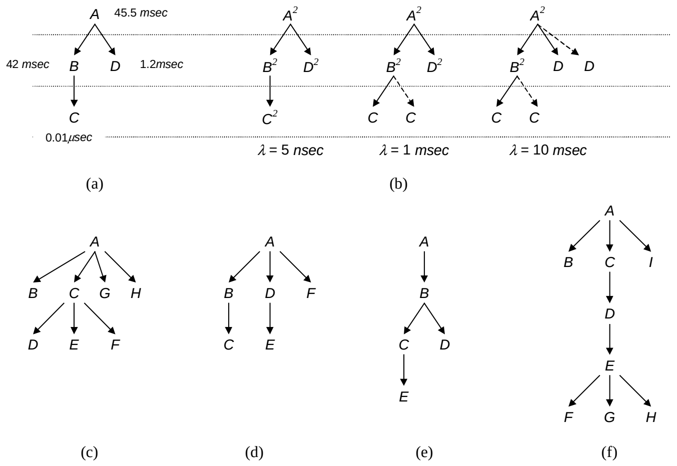


Fig. 4.1. (a) CG of Sort (b) Transformed programs with different error detection latency constraints in Sort (c) CG of MPEG (d) CG of LZW (e) CG of FFT (f) CG of Huff

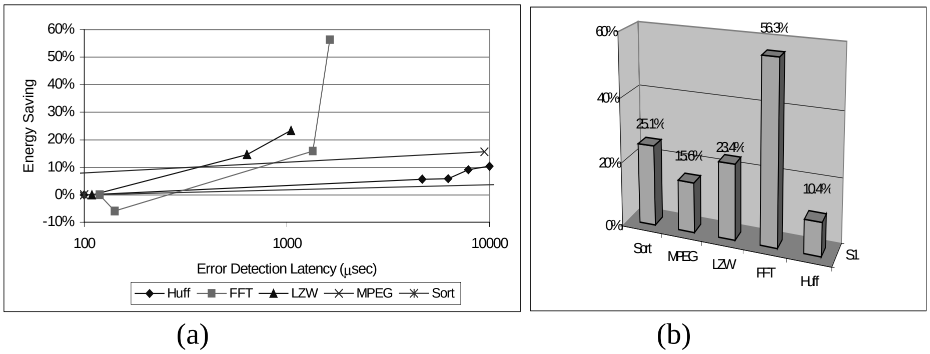


Fig. 4.2. (a) Energy savings with different values of error detection latency. (b) Maximum energy savings with the longest error detection latency.

Acknowledgements

This work was supported in part by the National Aeronautics and Space Administration and administered through the Jet Propulsion Laboratory, California Institute of Technology under Contract No. 1216777.

References

[1] Shirvani, P. and et al, “Software-Implemented Hardware Fault Tolerance Experiments: COTS in Space,” *Proc. Int’l Conf. on Dependable Systems and Networks, Fast Abstracts*, pp.B56-7, New York, NY, 2000.

[2] Lu, D.J., “Watchdog Processor and Sturctural integrity Checking,” *IEEE Trans. on Computers*, vol. C-31, No. 7, pp.681-685, July 1982.

[3] Oh, N., P. Shirvani, and E.J. McCluskey, “Control Flow Checking by Software Signatures,” *IEEE Trans. on Reliability to appear in Dec. 2001*

[4] Holm, J.G. and P. Banerjee, “Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions,” *Proc. Int’l Conference on Parallel Processing (ICCP)*, pp.192-195, 1992.

[5] Oh, N., P. Shirvani, and E.J. McCluskey, “Error Detection by Duplicated Instructions in Super-scalar Processors,” *IEEE Trans. on Reliability to appear in Dec. 2001*

[6] Brooks, D., V. Tiwari and M. Martonosi, “*Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,*”. Proc. of the 27th International Symposium on Computer Architecture, pp.83 –94, 2000.

[7] “A benchmark site on World Wide Web,” *http://www.netlib.org/benchweb/*

[8] M. W. Hall, and et al, “Maximizing Multiprocessor Performance with the SUIF Compiler,” *IEEE Computer*, vol. 29, pp.84-89, December 1996.