Software-Based Fault Tolerance in Computer Vision

Chen-Han Ho
CS766 Computer Vision
Final Project Report
Professor Vikas Singh

1. Introduction

Manufacturing and process scaling are providing significant challenges in producing reliable transistors for future technologies. Many academic experts, industry consortia and research panels have warned that future generations of silicon technology are likely to be significantly less reliable [11]. A recent Computing Community Consortium Visioning Study [1] conclude that handling reliability will probably become a first-order constraint.

To address this issue, modern processors often designed with architectural overheads, including architectural checkpoints, modular redundancy, and conservative design constraints. While these techniques provide error tolerant hardware, their power and energy efficiency is degraded. Furthermore, the problem exacerbates with technology scaling because the transistor efficiency does not double every generation. In mobile domain where the power and energy efficiency is critical, efforts have to be made to tackle both reliable and efficiency problems.

Stochastic optimization [9] and idempotent processing [3] are two fault-recovery approaches that allows the software to help mitigate the burden of hardware in reliability. Stochastic optimization recast applications to optimization problem and make application error tolerant. Idempotent processing leverages compiler to construct idempotent regions, and architectural mechanism to restart region execution whenever a failure happens. Comparing with pure algorithmic approach as stochastic optimization, the idempotent processing relies hardware to detect faults and restart execution. In addition, idempotent processing does not require error-free execution in the control phase of applications. In this project, we seek to understand the use of idempotent processing in computer vision, and compare it with stochastic optimization to explore the design tradeoffs. The remainder of this report is organized as follows: Section 2 gives a background of application robustification using stochastic optimization. Section 3 describes idempotent processing, and its architectural implications. Section 4 and 5 describe our evaluation and conclusion.

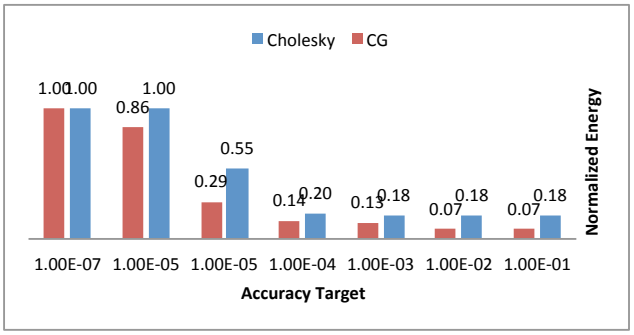

Figure 2. Normalized energy at different failure rate

2. Stochastic Optimization

In [9] Sloan et al. proposed to recast applications to numerical optimization problem for robustness. Assuming the solution of original problem is a vector $x*$, a cost function f is defined such that the minimum of f is attained at $x*$. Thus, we can transform an application from its original implementation to an error-tolerant implementation. The primary optimization engine used is gradient descent, which converges to a local optimum as long as the step sizes are chosen carefully. The search strategy used was conjugate gradient for most of the kernels that were evaluated in the paper. Conjugate gradient allows efficient generations of conjugate directions, and guarantees quick convergence. The result showed that the stochastic optimization can produce good quality results for certain application, and also reduced the energy by voltage scaling in least squares problem. However, in some cases the result was not that desirable. For example, the success rate of bipartite graph matching drops quickly as the failure rate increase (below 50% at 5% of failure). Improvements are made in the paper to enhance the success rate of error-tolerant bipartite graph matching. The enhancement method includes preconditioning, momentum, alternate step size scaling , and annealing. The success rate was improved to 100%.

Several limitations are observed in stochastic optimization. First, the algorithmic approach can only tolerant hard-

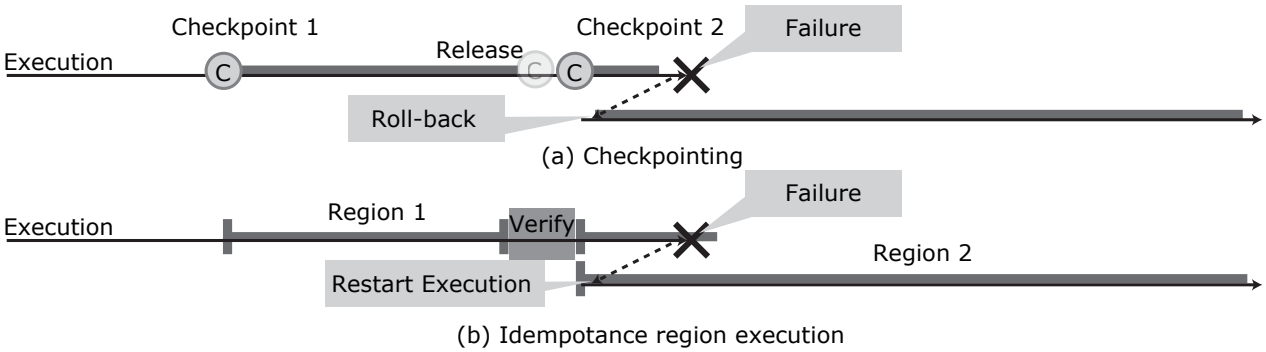

Figure 1. Checkpointing and idempotent execution

ware failures in specific phases of execution. The control phase cannot be erroneous, and it may be difficult to separate control and data phase for complex applications. Second, the iterative method may result in consuming more energy than original implementation, because the instruction executed may be up to 1,000X more. The above limitations may be solved (but with overheads) in hardware or software. For example, checkpointing or other hardware fault tolerant approaches can be applied whenever the application execution is in control phase.

The accuracy of transformed application highly depends on the solver. For example, the conjugate gradient solver guarantees most n iterations to solve $Ax = B$ when n is the dimension of x. However, conjugate gradient solver cannot achieve an accuracy higher than 10^{-5}, where the SVD and QR decomposition can achieve 10^{-2} accuracy. (The accuracy here is the relative error to the non-error tolerant implementation of same application.) Besides speed and accuracy, the solvers may not be universal thus the generality have to be considered when implementing the application.

Last, Figure 2 shows the normalized energy at different failure rates for least squares problem. The numbers are extract from [9], and the energy numbers are normalized to Cholesky implementation at 10^{-7} accuracy. In the result, the voltage and iteration of the solver are scaled at the same time to achieve best result. The best saving observed are 93% and 82% at 0.1 relative error, for CG and Cholesky solver respectively.

3. Idempotent Processing in Computer Vision

Idempotent processing [3, 4, 8] leverages the principle of idempotence to break programs into regions of code than can be recovered through simple re-execution. Figure 1 shows an example execution of idempotent processing. Idempotence is a mathematical property which guarantees that any region of code (a sequence of instructions) can be freely re-executed, even after partial execution, and still produce the same result. With idempotent regions, the execution can be implicitly checkpointed. Figure 1(a) shows a reference execution of hardware checkpointing. The redundant hardware saves the state of execution, which can be retrieved whenever failure happens. In Figure 1(b) shows corresponding idempotent execution. Whenever a failure happens, the hardware detects the failure and restart the execution from the beginning of the region. The hardware must ensure that the execution is free of failure before start execute next idempotent region and all side-effects of failure are appropriately contained. This is shown as a dark "verify" box in the execution. Idempotent regions are identified in the assembly code level. A region of code is idempotent if it does not overwrite its inputs. For a variable in a code region, the idempotence breaks whenever there is a read-after-write dependence with no prior write in the same region. While idempotent regions can be identified in the program binary generated by normal compilation, idempotent-aware compilation helps to control the size of idempotent regions through different region construction algorithm. Since there are a limited number of storage resources (e.g. registers) in hardware, the compiler re-uses the resources in normal compilation. However, this sometimes breaks idempotence by overwriting the same storage resource and can be avoided in an idempotent aware compilation. Idempotent compilation often accompanied with the increase of total number of instructions, because compiler will not perform some of the common optimizations to reduce the use of resources. Overall, the overhead of recovery from a failure in idempotent processing comes from i) the time of re-execution, ii) the additional instructions added to preserve the idempotence, and iii) the hardware verification phase.

The size of idempotent region impacts the time of re-execution. In an error-prone system, we may want to construct smaller idempotent regions to reduce the re-execution time. As stochastic optimization, idempotent processing recovers failure in software. This eliminates the need of preserving execution states in hardware. Compared to stochastic optimization, idempotent processing requires hardware to detect failures, verifying idempotent region execution,

and restart the execution when failure happens. This requires modification in hardware, but not necessarily increases the complexity and power consumption in hardware. Idempotence may be used in other places than fault tolerance. De Kruijf et al. proposed to use idempotent processing to simplify the microprocessor design [3]. In their case, the idempotence is used to recover architectural exceptions as branch mis-predictions and page faults.

In contrast to stochastic optimization, the re-execution in idempotent processing always produce identical result as error-free execution. This enables the use of idempotent processing in many different algorithms. For example, the Bipartite Graph Matching is often considered not a error tolerant algorithm. Although it can be potentially error tolerant through stochastic transformation, no algorithmic modification is required and 100% of accuracy can be preserved with idempotent processing.

4. Evaluation

4.1. Methodology

To understand the idempotence in computer vision applications, we use a subset of kernels in VLFeat [2] open source library: Agglomerative Information Bottleneck (AIB), Maximally Stable Extremal Regions (MSER), Scale Invariant Feature Transform (SIFT), vector comparison (VEC), and image convolution (CONV). AIB implements the algorithm describe in [10], which includes traversing customized data structure and merging thorough binary tree. MSER algorithm is described in [7]. It includes walk through pixels and filtering. SIFT implements the algorithm in [5], which also includes filtering, sampling and data movement. The above three are full applications, and CONV and VEC are samll kernels that only perform one task. Table 1 characterize these kernels in terms of the size idempotent region construed by the compiler. The average idempotent region size is shown in the number of instructions. Among the kernels, VEC has the largest region sizes since its memory access pattern is highly regular. The second row in Table 1 shows the input set size. We choose a smaller input set to control the simulation time. All fault injected experiments finished within 24 hours.

Kernel	AIB	MSER	SIFT	VEC	CONV
Region size	250	12	27	1056	95
input set	3*10	324*223		500*1000	256*256

Table 1. Application characterization

We use idempotent compiler framework developed by De Kruijf et al. to generate idempotent version of programs. We build an instrumentation tool with Pin [6] to probabilistically insert failures and jump instructions, which emulate the behavior of re-execution. We assume that failure can happen in every instruction, and the probability of hardware failure is independent for each instruction. As a result, the failures has a Poisson distribution. The failures are bit-flipped result that can be written into the same destination register or memory. The instrumented binary is executed on a Intel Core2 Duo E7200 processor with RedHat 6 Enterprise Linux.

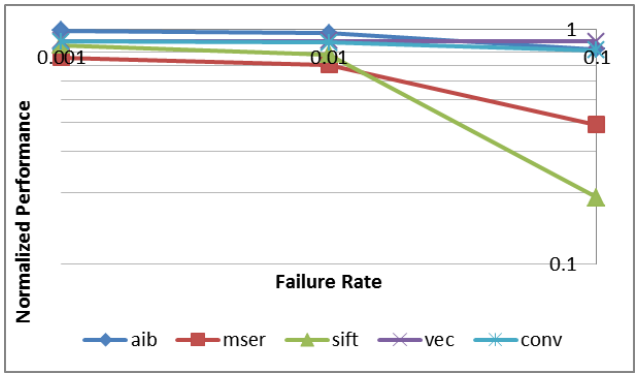

Figure 3. Normalized performance compared to baseline implementation

4.2. Results

The overhead of idempotent processing is twofold. First, idempotent processing requires compiler to generate implicit software checkpoints, which may results in more instructions. Second, idempotent processing utilizes re-execution to achieve 100% correct results. Figure 3 shows the normalized performance of kernels in different failure rate. The performance of each kernel is normalized to the execution time of its baseline binary with no idempotence. The execution of original binary is also instrumented with the instrumentation tool but with zero failure rate. At 0.001 failure rate, the idempotent version of kernels achieves 75% to 98% performance of baseline. In an extremely high failure rate of 0.1, two of the five kernels still performs similar. However, SIFT and MSER drops quickly to 5X and 2.5X slower, and VEC performs 16X slower. Compared to 10X to 1000X increase of instructions in stochastic approach, this is relatively smaller. In addition, the idempotent region size can be optimized to achieve a better result in highly error prone system.

To understand the potential benefit in energy, we show the normalized energy of each kernel in Figure 4. We use the same voltage scaling assumption for FPUs in [9], and the energy is normalized to the baseline binary. When scaling the voltage down (and hence the error rate increases), we observed two of the five kernels uses lower energy compared to the baseline. The worse case SIFT shows a 3.6X increase in energy at 0.1 failure rate. Compared to the least squares stochastic implementation, idempotent processing always provides 100% accurate result but may increase the

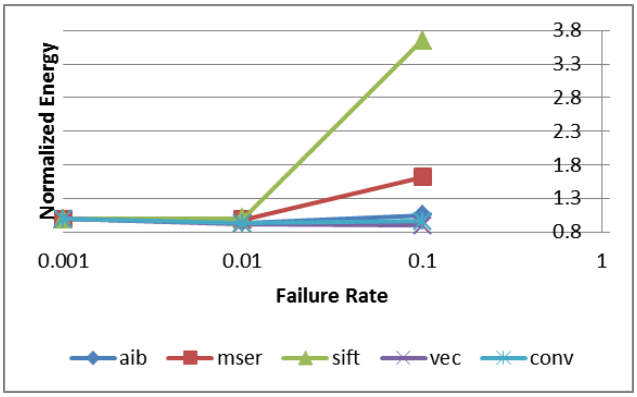

Figure 4. Normalized energy compared to baseline implementation

energy consumption because the increased execution time, while the stochastic approach can always reduce the energy by lowering the accuracy target. Idempotent processor itself may be more energy efficient, but in this study, we only consider the effect of voltage scaling.

4.3. Challenges, Limitations and Future Work

We address several challenges and limitations in this project. First, current instrumentation tool inserts one jump instruction after every instruction that is not a idempotent region marker. This incurs high overhead and can be improvement to make the instrumentation tool more efficient. Second, current idempotent compiler has optimization option for speed, however, it is built for branch prediction. An optimization for fault tolerance is necessary for future analysis.

The assumption of voltage scaling and failures in [9] are relatively high in fault tolerance study, this may because they assume a stochastic processor which allows more failures. Besides, some important data is not presented in Sloan paper, including the execution time of applications. The energy result in the paper is difficult to be compared with since both the number of iterations and voltage is changed. A more sound comparison can be made if only one of the variable is change in experiment. In all, re-implementing stochastic optimization algorithms and using the instrumentation tool to perform the experiments would be the next step.

5. Conclusion

Reliability is an important constraint for future systems and consensus is emerging around software fault recovery to mitigate the burden in hardware. This project provides a quantitative study of the use of idempotent processing in computer vision applications. The hardware only has to guarantee detection, and recovery is achieved through simple re-execution in software. In a system with 0.001 failure rate, idempotent processing provides 100% execution accuracy with 75% to 98% lower performance. Compared to stochastic optimization, the idempotent processing has better accuracy with no programming effort. However, stochastic optimization can have better energy saving by lowering the accuracy target.

We thank Marc de Kruijf for his help in idempotent compiler, professor Karu Sankaralingam and professor Vikas singh for their advices on the project.

References

[1] Ccc visioning study on cross-layer reliability, http://www.relxlayer.org/. 1

[2] The vlfeat open source library, http://www.vlfeat.org. 3

[3] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 140–151, New York, NY, USA, 2011. ACM. 1, 2, 3

[4] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of 33rd International Conference on Programming Language Design and Implementation "'(PLDI)"'*, 2012. 2

[5] D. Lowe. Object recognition from local scale-invariant features. pages 1150–1157, 1999. 3

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200. 3

[7] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *In British Machine Vision Conference*, pages 384–393, 2002. 3

[8] J. Menon, M. de Kruijf, and K. Sankaralingam. igpu: Exception support and speculative execution on gpus. In *Proceedings of 39th International Symposium on Computer Architecture "'(ISCA)"'*, 2012. 2

[9] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 161 –170, 28 2010-july 1 2010. 1, 2, 3, 4

[10] N. Slonim and N. Tishby. Agglomerative information bottleneck. pages 617–623. MIT Press, 1999. 3

[11] A. W. Strong, E. Y. Wu, R.-P. Vollertsen, J. Sune, G. L. Rosa, T. D. Sullivan, S. E. Rauch, and III. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley-IEEE Press. 1