Analyzing the Effects of Compiler Optimizations on Application Reliability

Melina Demertzi
Computer Science Department
University of Southern California
demertzi@usc.edu

Murali Annavaram
Electrical Engineering Deparment
University of Southern California
annavara@usc.edu

Mary Hall
School of Computing
University of Utah
mhall@cs.utah.edu

Abstract

As transistor sizes decrease, transient faults are becoming a significant concern for processor designers. A rich body of research has focused on ways to estimate the vulnerability of systems to transient errors and on techniques to reduce their sensitivity to soft errors. In this research, we analyze how compiler optimizations impact the expected number of failures during the execution of an application. Typically, optimizations have two effects. First, they increase structures occupancies by allowing more instructions in flight, which in turn increases their susceptibility to soft errors. Additionally, they decrease execution time, decreasing the time during which the application is exposed to transient errors. In particular, we focus on how optimizations impact occupancies in three processor structures, namely the Reorder Buffer, the Instruction Fetch Queue and the Load Store Queue. We explain the interplay between compiler and reliability by studying the changes in the code made by the compiler and the resulting responses at the microarchitectural level. Results from this research allow us to make decisions to keep an application within its performance goals and its vulnerability during its runtime within a well defined FIT target.

1 Introduction

Silicon technology scaling and reduced supply voltage have severe repercussions on processor reliability. The reduced amount of charge held in an SRAM cell makes the processor highly susceptible to transient errors. The new mechanisms proposed to reduce leakage current, such as operation at near-threshold voltages [6] as well as the aggressive power optimization techniques that further reduce the critical charge make the devices even more susceptible. Thus, soft errors have become of significant concern to the industry [2].

While soft errors can affect any block within a processor, storage structures such as the Instruction Fetch Queue, the Load/Store Queue, and the Reorder Buffer are particularly vulnerable to soft errors. The probability of an application encountering a soft error during its execution increases if the instructions spend more time in these susceptible structures. Much research has been done to show how this susceptibility to soft errors varies during the execution of a program and also varies from one program to another [19, 1, 16, 15, 17]. But fundamentally an application's occupancy in these structures is determined by how a compiler generates code. To date limited research exists on how compiler optimizations impact the susceptibility of microarchitectural structures to soft errors. The research presented in this paper focuses on analyzing the impact of compiler optimizations on an application's reliability.

Compiler optimizations indirectly impact reliability in multiple ways by affecting the number of instructions in-flight and thus the occupancies of the processor's storage structures. Examples include dead code elimination and loop-unrolling. Both of these optimizations bring more useful instructions in the processor where they are susceptible to soft errors. On the other hand, there are code optimizations that reduce an application's susceptibility. For instance, moving a dependent instruction away from a load reduces the waiting time, especially if the load misses in cache and thus reduces the susceptibility of the instruction to a particle strike.

Additionally, compilers affect an application's reliability by decreasing the code predictability and thereby causing abrupt variations in structures occupancies. For example, memory reference elimination and their replacement with registers eliminates the most predictable memory references and results in higher cache miss rates. Waiting on a cache miss, leads to higher structures occupancies. Similarly, loop unrolling eliminates the more easily predictable branch instructions leaving fewer and not as predictable branch instructions. A branch misprediction leads to pipeline flushes and empty storage structures. Hence, these sudden variations in structure occupancies results in large variance in the application susceptibility.

Finally, faster execution of optimized code reduces the

amount of time an application is exposed to soft errors. Hence, even if a structure has higher susceptibility the duration of the susceptibility may be reduced. This may result in a net reduction in the probability of a soft error strike during an application run.

As demonstrated amply above, one can not ignore the role of compilers in affecting an application's susceptibility to soft errors during execution. The goal of this research is to analyze the relationship between the compiler optimizations and the system's reliability and provide insights into how microarchitecture structure occupancies change with compiler optimizations. We show that overall optimizations help reduce the expected number of failures seen by an application during its execution. However, we also show that optimizations lead to unpredictable variations in the expected number of failures seen during a short time interval. These results are valuable not only for application and compiler developers but also to hardware designers. For instance, hardware designers may provide error protection to account for the worst case expected failure rate. Hence, optimized code may in fact lead to higher protection overhead.

2 Related Work

It has been shown that the susceptibility of an application to transient errors varies drastically across applications and within different phases of the same application [13, 19]. Jones, *et al.* [11] have made an initial attempt to evaluate the effect of the compiler optimizations on reliability, using AVF metric. They quantified the tradeoffs between performance and reliability when compiler optimizations are applied while running MiBench benchmarks [9] on a Xscale processor.They primarily focused on quantifying the impact of optimizations on AVF without analyzing the underlying code behavior. They reported the optimization effect on performance as "positive" or "negative" but did not elaborate or studied the reasons behind this behavior. The primary contribution of our work is to provide a more detailed insight into how code fragments generated by the compiler interact with microarchitecture to impact the system's vulnerability to transient errors.

Cook, *et al.* [4] characterize the instruction-level derating of the SPEC CPU2000 integer benchmarks and classify them into six categories. The term *derating* is used to indicate the cases where the use of incorrect data can still lead to correct results, as opposed to the term *masking* where the incorrect data is never accessed or utilized. The authors use the Simplescalar simulation infrastructure to inject errors in the benchmarks when they are compiled with the -O0, -O2 and -O3 optimization levels. The classification and analysis of the derating factor was done at the instruction level. On the other hand, our work looks into the micro-architectural behavior that results from the various optimization levels applied and its effect on overall reliability.

Mukherjee, *et al.* [16] showed that not every bit flip results in a *visible error* at the output of a program. Hence, they used the concept of AVF as a way to more accurately estimate the FIT. Their methodology was extended in following papers to include SRAM-based structures, such as a store buffer [3] and cache [21, 20]. Efforts to provide online and light-weight ways for AVF estimation are presented in [19, 5, 12, 13]. AVF provides instantaneous probability that is not correlated to the entire execution time of the application. Thus, it cannot answer questions such as what is the relative reliability level of an application that is twice as vulnerable but also twice as fast as a second one. A number of different metrics, such as Mean-Work-to-Failure (MWTF) [18] and Mean-Instructions-to-Failure (MITF) [8] have been proposed to capture this reliability/performance trade-off. In our study, we use the *expected number of failures during execution* as an alternative reliability metric for various reasons. Since the optimized and unoptimized versions vary drastically in terms of instruction count and execution time, any instantaneous metric would not be appropriate. Moreover, it would be difficult to isolate points in execution where equivalent execution takes place. Hence, we use the expected number of failures as our metric.

3 Application Reliability Metric

Any metric for measuring application's vulnerability to soft errors must allow us to compare various optimized versions of the code taking into account the execution time differences. In this paper, we use *expected number of failures during the application's execution* (EF). In the context of this paper, we use the term application reliability to describe the susceptibility of an application to transient errors. Thus, an application run is more reliable if a particle strike is less probable to produce a visible failure at the output. The EF metric is an accurate metric measuring the overall application reliability as it represents the probability that an application correctly finishes execution or fails with an error, which is the main concern of end-users. EF is computed using Equation 1 shown below.

$$EF = IntrinsicFIT \times AVF \times ExecutionTime \quad (1)$$

In this equation the intrinsic FIT rate represents the probability that a particle strike results in a bit flip, while the AVF corresponds to the probability that this bit flip results in a visible error. Finally, the execution time in this equation accounts for the duration of the susceptibility; the longer an application runs, the longer it is susceptible to soft error strikes. EF metric is additive and hence EF in each structure during a program execution can be added to compute the total number of expected failures encountered by the application.

We focus on computing the expected number of failures an application is likely to encounter in three SRAM-based processor structures, namely *Instruction Store Queue* (ISQ), *ReOrder Buffer* (ROB) and *Load/Store Queue* (LSQ). We deliberately chose one structure from the processor front-end (ISQ), one structure from the processor back-end (ROB) and one structure from the memory pipeline (LSQ). The three structures used in this study represent the most critical structures that hold instruction/data state and are a good proxy for the overall failures likely to be encountered in the processor.

Intrinsic FIT computation: The intrinsic failure rate of a device is usually defined by the number of failures in 1 billion hours of operation (FIT). For SRAM structures built using 65 nm technology node operating at 3GHz, the intrinsic FIT rate is approximately 1150 FITs per Megabit of SRAM [15]. We assume that our underlying hardware uses these technology parameters. We compute the intrinsic FIT rate for the structures of interest (ROB, ISQ, LSQ) using Equation 2 to estimate the number of failures in one minute.

$$IntrinsicFIT = size_{inMB} \times \frac{1150}{10^9 \times 60} \qquad (2)$$

AVF computation: The next challenge in our study is to compute the AVF of the three SRAM structures. In our study we were interested in measuring EF continuously as a program executes and hence we opted for an online AVF estimation mechanism. Walcott, *et al.* [19] proposed a very simple mechanism to compute AVF online using only the occupancy of microarchitecture structures. Using extensive empirical evaluations they provided a set of equations to compute AVF for a 8-wide issue out-of-order processor. The processor has 8 wide fetch/decode and commit capability with 128 entry ROB, 64KB L1 cache and 512KB L2 cache. Table 1 shows the equations for computing the AVF of three structure of interest to this study. The constants shown in the equation are empirical constants for a particular microarchitecture and are produced after a rigorous statistical analysis on the AVF behavior of a modern processor. These equations reveal a high correlation between the AVF level of the structures and their respective occupancies. Since AVF is dependent on the number of bits stored in a structure, it should be intuitively obvious that a structure with higher occupancy has a larger probability to include bits that are needed for architecturally correct execution. Thus high occupancy is related to high AVF values.

Execution Time: We compute the execution time as the number of simulation cycles reported by our simulation infrastructure. We simulated 5 SPEC 2000 benchmarks [10] to completion using the train set as input. Simulation infrastructure and choice of benchmarks are discussed in more detail in Section 4. The choice of simulating the benchmarks to completion, instead of using sampling methods to simulate only representative pieces of the application, has been influenced by the subject of our research. We needed to simulate multiple versions of the same application and compare how EF varies across different optimized versions. Based on the optimization level applied, the resulting binaries differ drastically from each other making it hard to pinpoint equivalent points during execution. Therefore, it would be difficult to fast-forward to points in time where equivalent computations take place or knowing for how long to execute. Following the same rationale, the representative SimPoints would be different for the different code versions and the comparison of EF across binaries would be unfair. In short, by simulating the benchmarks to completion, we ensure that all versions of code perform equivalent functions despite their different execution lengths and that any comparison between them is fair. The train set input was chosen to balance prohibitively long simulation times and representative code size. Each benchmark simulated with the train set input executes tens of billions of instructions providing a realistic and representative outcome, while allowing us to simulate a large number of benchmarks. The reference input set would lead to prohibitively long simulation times, particularly given that we are simulating a detailed out-of-order processor model.

Struct.	Equation
ISQ	$-0.90 + (4.83I_o) + (0.74L_o) + (0.81R_o)$
ROB	$-3.06 + (0.73R_o)$
LSQ	$-1.23 + (1.14L_o)$

Table 1. AVF estimation equations as proposed in [19]. R_o is ROB occupancy, I_o is ISQ occupancy and L_o is LSQ occupancy.

4 Experimental Setup

We use the methodology outlined in Section 3 to measure how compiler optimizations impact EF during execution. We selected 5 of the 12 CPU2K integer benchmarks. Simulating multiple optimized versions of benchmarks to completion when each run takes several days is limited by our computational infrastructure. We compiled the benchmarks with gcc (version 3.4.6) [7]. While users of gcc can select individual optimizations to apply from the list of roughly 200 available optimizations and their associated parameters, the simplest and most common way of producing optimized code by end-users is by turning on the "optimization level" (-O1, -O2 ... -On). The first optimization flag (-O1) turns on 28 optimizations that reduce execution time and code size by improving register allocation and a few branch optimizations for control flow simplification. The second optimization level (-O2), turns on 32 more sophisticated optimizations such as instruction scheduling, inlining when it does not increase code size, global common subex-

pression elimination. Finally, the last optimization flag (-O3) turns on 5 additional aggressive optimizations that may increase code size, primarily to further simplify control flow and improve ILP: inlining, unrolling and "vectorization" for SSE-3 SIMD instructions.

We run the SPEC2K benchmarks [10] and keep track of the structure occupancies to compute the *expected number of failures during execution* using Equation 1. The constants in the AVF computation as shown in Table 1 are dependent on the microarchitecture of the system on which the analysis was performed. It is likely that these values will differ if the microarchitecture varies. Hence, we selected a simulation based infrastructure to configure precisely a processor as described in [19] for which the AVF equations are computed. We also log the simulation time required for the execution of the instructions. For our experiments, we use *zesto* [14], a cycle accurate simulator of the x86 architecture, modified to estimate the expected number of failures of the processor's hardware structures.

We study the reliability behavior of our benchmarks with the SPEC CPU2K train set as input. Each one of these benchmarks includes tens of billions of instructions and by running them to completion we ensure that our results are meaningful and representative. We compute the AVF values for the hardware structures every 10,000 instructions, i.e. every 10,000 instructions, we read the structures' respective occupancies to compute the AVF using the equations shown in Table 1. We then measure the elapsed time (simulation cycles) for 10, 000 instruction completion and compute the number of expected failures for every 10,000 instruction execution window.

5 Experimental Results

In this section, present results from our experimental study. We first created four versions of our five benchmarks, each compiled with -O0, -O1, -O2, -O3 optimization levels. We then simulated all four optimized versions of each benchmark to completion on *zesto*. We analyze snippets of assembly code to understand how optimizations impact EF metric for all our benchmark runs. We first provide an overview of the experimental results and note interesting trends and behaviors of how the EF metric varies across the three structures in response to the compiler optimizations. Then, we explain the measured results by tying the results to interesting events at the microarchitectural level, such as cache misses and branch mispredictions. We provide answers to fundamental questions such as: How do optimization affect the number of cache misses and what effect does this have on the number of failures encountered during an application run? When needed we provide results for all benchmarks across all optimization levels. However, in the interest of space and clarity we provide a comparative view of the unoptimized code (-O0) and the optimized version that provides the best performance in terms of execution time (-Ox).

	exec. cycles (E+10)	instructions (E+10)	ROB EF (E-10)	ISQ EF (E-10)	LSQ EF (E-10)
O0	2.215	1.540	1.90	41.5	0.20
O1	1.405	0.999	1.59	32.1	0.13
O2	1.410	1.072	1.54	30.9	0.12
O3	1.015	0.716	1.11	22.0	0.08

(a) *mcf*

	exec. cycles (E+10)	instructions (E+10)	ROB EF (E-10)	ISQ EF (E-10)	LSQ EF (E-10)
O0	2.494	1.382	1.09	29.9	0.11
O1	1.562	0.975	0.76	20.2	0.06
O2	1.650	0.985	0.67	19.6	0.05
O3	1.250	0.801	0.52	14.9	0.04

(b) *parser*

	exec. cycles (E+10)	instructions (E+10)	ROB EF (E-10)	ISQ EF (E-10)	LSQ EF (E-10)
O0	2.748	1.863	0.52	18.2	0.05
O1	2.180	1.483	0.40	15.3	0.04
O2	2.246	1.527	0.35	14.3	0.03
O3	2.260	1.506	0.35	14.0	0.03

(c) *vortex*

	exec. cycles (E+10)	instructions (E+10)	ROB EF (E-10)	ISQ EF (E-10)	LSQ EF (E-10)
O0	14.49	10.35	7.21	217	0.76
O1	6.125	4.630	0.55	46.4	0.04
O2	5.740	4.309	0.44	44.1	0.04
O3	6.387	4.727	0.73	52.1	0.07

(d) *bzip2*

	exec. cycles (E+10)	instructions (E+10)	ROB EF (E-10)	ISQ EF (E-10)	LSQ EF (E-10)
O0	7.928	5.503	4.24	82.9	0.32
O1	5.231	3.840	1.91	53.3	0.13
O2	5.227	3.847	2.60	54.0	0.17
O3	5.176	3.811	1.42	45.8	0.11

(e) *gzip*

Figure 1. Performance and reliability characteristics of benchmarks

The five tables in Figure 1 present the performance and EF characteristics of the five selected SPEC2K integer benchmarks. The first two columns present the execution time of the application in terms of clock cycles and the number of executed instructions. The last three columns present the expected number of failures during the execution of the application (the EF metric) for the ROB, ISQ and LSQ respectively. As expected, optimizations alter dramatically the execution time of the benchmarks causing an improvement of as much as $2.6\times$ in the case of *bzip2*. It

should not be surprising to note that not all benchmarks react equally well to the applied optimizations and that increasing optimization levels does not always improve performance. Even in our five benchmarks there are six cases when an increased optimization level actually hurt performance. For parser and mcf -O2 has worse performance than -O1, for bzip2 -O3 performs worse than -O2 and in vortex -O2 and -O3 are both worse than -O1. One strong observation is worth making: never is the case that an unoptimized code executes fewer instructions or completes in fewer cycles than an optimized code.

Now we discuss the reliability implications of optimizations. The EF metric of the three structures shows that reliability also improves with optimizations. Lower EF values imply fewer failures. Recall from Equation 1 that the EF value depends on the execution time. If execution time reduces drastically, as is the case when we move from unoptimized to some version of optimized code, then EF also drops dramatically. In fact our measurements show that execution time has stronger influence than AVF on EF value. AVF values do vary with optimizations but their contribution to EF computation is overshadowed by execution time changes.

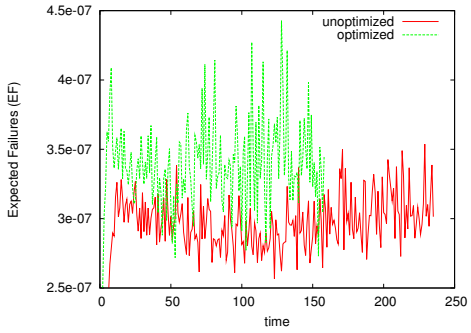

Figure 2. Time variation in EF during the execution of *mcf*.

To demonstrate how AVF and execution time contribute to the EF metric, in Figure 2 we plot the time varying behavior of EF at a much finer granularity for the mcf benchmark. We plot the expected number of failures at a 10,000 instructions granularity for 25 million instructions. There are three important observations to be made from Figure 2. First, for most points in time, the unoptimized code is actually more resilient, i.e. the instantaneous EF value of the unoptimized code is consistently lower than for the optimized code. However, the unoptimized code has higher overall EF because it runs for a longer period of time, as is obvious by the long time tail of the unoptimized code in the graph. This tail is the main reason for why unoptimized code is more susceptible to soft errors. Finally, the optimized code exhibits a much more variable and thus unpredictable behavior in its EF metric in a given time window. This unpredictability is highly undesirable for hardware architects and designers because it obliges them to account for the worst case scenario while considering protection mechanisms against soft errors. For example, a high number of expected failures even for a short period of time, might preclude the use of simple protection mechanisms such as the inclusion of parity bits and require more complex and expensive strategies such as ECC. Conversely, the smoother and more predictable behavior of the unoptimized code allows the use of parity even if overall EF is higher. In Section 6, we describe a hypothetical system design that exploits the lower variance in EF of unoptimized code to reduce hardware cost while still staying within a maximum tolerable FIT value.

5.1 Cache behavior

Cache behavior determines to a large extent an application's performance and therefore the duration of time it occupies vulnerable SRAM structures. The importance of memory and cache behavior on the execution time has led to the development of many compiler optimizations to target the memory sub-system. Compilers strive to minimize memory access time by eliminating redundant accesses to memory. They eliminate memory accesses and replace them with shorter latency events, such as register copying and value propagation through registers. An example of redundant memory accesses elimination can be seen in Table 2, where we see the assembly code generated for the C statement `return(cost-tail->potential+head_potential);` taken from the *mcf* benchmark. Table 2:Unoptimized presents the unoptimized assembly code, while Table 2:Optimized presents the assembly code generated by the -O1 flag. First, we observe that the unoptimized code is longer than the optimized one and that it includes more data transfers (8 mov instructions instead of 5). The unoptimized code performs the addition of 3 numbers by loading the first two operands in registers, adding them and storing the result in a third register (Table 2:Unoptimized lines 05-08). Then, loads the third operand, adds it to the result of the previous operation and returns. The optimized version avoids the step of storing the intermediate result (Table 2:Optimized lines 05-08), leading to a more compact and efficient code which avoids one memory access and eliminates one data dependence. As a result, the optimized code includes only 5 move instructions instead of 8. This lower number of memory accesses in the optimized binary's execution is consistent across all benchmarks. Figure 3(a) plots the total number of memory references (cache accesses) in optimized code normalized to the unoptimized version of the code. As can be seen, the optimized code on average has 45% fewer memory

01:	pushl	%ebp
02:	movl	%esp, %ebp
03:	movlr	08(%ebp),%ecx
04:	movlr	08(%ebp),%eax
05:	movlr	(%eax),%eax
06:	movlr	02c(%eax),%edx
07:	movlr	010(%ecx),%eax
08:	subl	%edx, %eax
09:	movlr	08(%ebp),%edx
10:	movlr	04(%edx),%edx
11:	addl	02c(%edx),%eax
12:	leave	
13:	ret	

Unoptimized

01:	pushl	%ebp
02:	movl	%esp, %ebp
03:	movlr	08(%ebp), %edx
04:	movlr	(%edx), %ecx
05:	movlr	010(%edx), %eax
06:	subl	02c(%ecx), %eax
07:	movlr	04(%edx), %edx
08:	addl	02c(%edx), %eax
09:	leave	
10:	ret	

Optimized

Table 2. Assembly code snippet of the bea_compute_red_cost() function of the *mcf* benchmark.

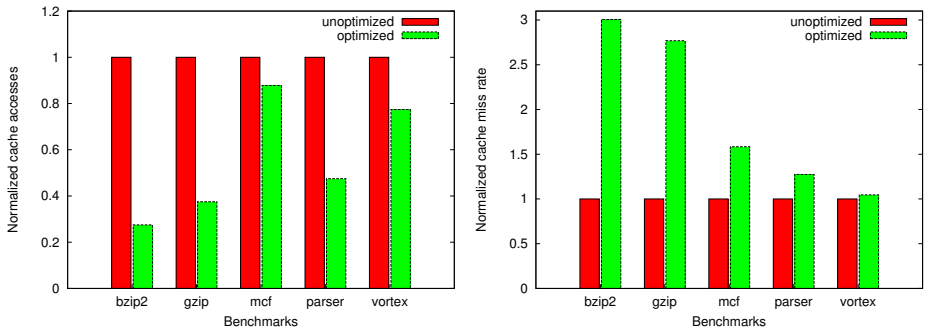

Figure 3. (a) L1 Data cache accesses (b) L1 Data cache miss rates

references.

After the removal of the redundant cache accesses by the compiler, the remaining accesses in the application are more likely to miss in cache. This leads to an increased cache miss rate for the optimized versions of the code, as can be seen in Figure 3(b). This fact is consistent across all our benchmarks.

One direct impact of reducing cache accesses with optimizations is that the average LSQ occupancy will be reduced. Hence, the expected number of LSQ failures during execution is lower for the optimized versions of the code as we can see for the mcf benchmark in Figure 4(a). Fewer memory instructions occupy fewer entries in the LSQ contributing to a lower occupancy and consequently to lower susceptibility to errors. Figure 4(b) presents the average LSQ occupancy during the execution of *mcf*. Moreover, the more efficient cache utilization lowers the dependencies between those LSQ entries and improves the time which each instruction occupies the LSQ entries.

The occupancy of the ROB is also impacted by the application's cache behavior, a fact manifested in its resulting EF value as seen in Figure 5(a). The increased number of cache misses affects the ROB occupancies in two ways. First, when a cache miss occurs hundreds or even thousands of cycles are needed for it to be serviced. During this period of time instructions continue being issued, filling up the ROB entries. If the cache miss is not serviced within a period of time, the ROB becomes full and the processor stalls until the instruction at the head of the ROB can retire. Therefore, a cache miss leads to higher ROB occupancies and as a result a higher susceptibility to transient errors. The increased number of cache misses and the short execution time in the optimized code result in the ROB operating more frequently at close-to-full state, leading to a higher average ROB occupancy. In its turn, this means a higher instantaneous EF metric during execution. On the other hand, the optimized code is more efficient in discovering higher degrees of instruction-level parallelism, allowing the simultaneous execution and retirement of multiple instructions. In this case, as soon as the cache miss is resolved, the ROB occupancy drops drastically as many instructions can retire and operates at low occupancy until the next cache miss. In conclusion, the cache behavior accounts for both higher average value and variation in the ROB occupancy. As depicted in Figure 5(b), this fact leads to higher variations in the ROB EF as well.

An interesting observation comes from the close relationship between the reliability of the ISQ and that of the

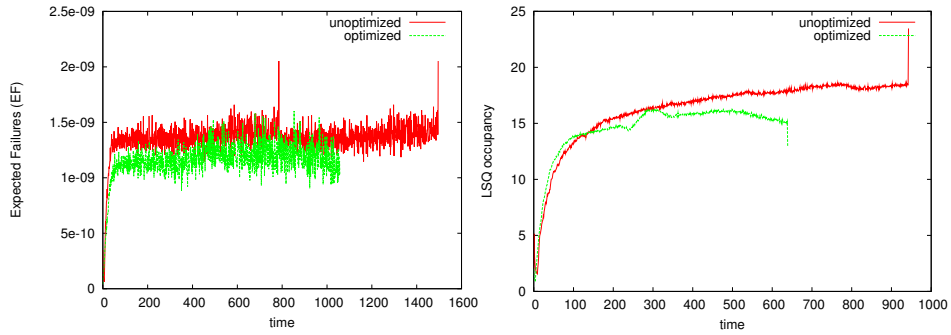

Figure 4. (a) LSQ EF for *mcf* (b) Average LSQ occupancy for *mcf*

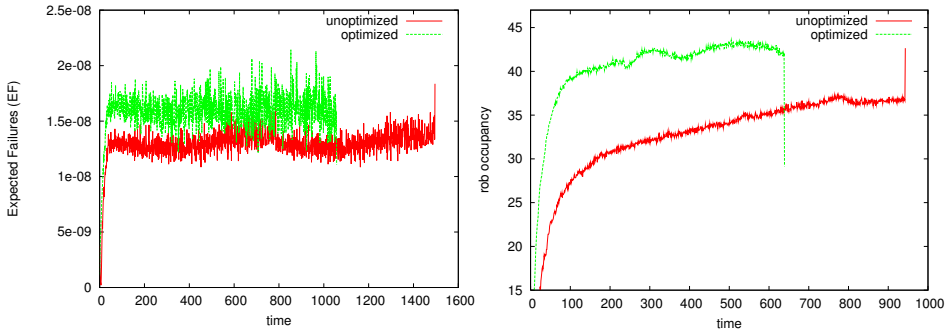

Figure 5. (a) ROB EF for *mcf* (b) Average ROB occupancy for *mcf*

ROB, a fact explained by the dependency between their relative functions. The ISQ acts as a buffer where the instructions coming from the instruction cache wait to be drained in the pipeline. If the back-end of the processor is efficient enough and can retire instructions at the same rate that the front-end fetches them, the ISQ will operate almost always empty, having an AVF factor of almost zero. However, such a balanced machine is rarely the case and long-latency operations such as cache misses result in a full ROB. While the ROB is full and no instructions retire, the ISQ starts filling and its susceptibility to errors increases rapidly. Thus, we can see that the reliability of the ISQ depends on the reliability of the ROB. Based on this observation, we can say that a reduction in the vulnerability of the ISQ can come from an increase in the efficiency of the processor's back-end. If instructions can retire as soon as possible and without spending a long time in the structures' entries, the expected failures of both the ROB and ISQ will be reduced. The most reliable processor will be a well-balanced machine that instructions can be committed at the same rate that they are issued.

The increased number of cache misses also affect the ISQ EF. The ISQ occupancy follows ROB occupancy trends, albeit with a delay because when the ROB is stalled, no instructions can enter the processor pipeline which will lead to ISQ stalls after a short delay. Therefore, the increased number of cache misses increases the average ISQ occupancy. Figure 6 verifies our claim by showing a higher average occupancy for the optimized code during the entire execution of the *mcf* benchmark. At the same time though, the more efficient cache use lowers the ISQ occupancy variation because the waiting time between an instruction being issued and executed is decreased. Thus, the instructions fetched at the ISQ do not reside in the structure for extended periods of time keeping the ISQ occupancy low. This fact and the shorter execution time results in fewer number of expected ISQ failures for optimized code.

5.2 Branch behavior

Another application characteristic that affects its performance is control flow. A delay in branch resolution can lead to stalls until the branch outcome is resolved. A mispredicted branch can lead to pipeline flushes and roll-back to a consistent state where execution on the correct path can resume. These two events have conflicting outcomes on the application's reliability.

In Figure 7 we can see that the optimized benchmarks have fewer committed branches and fewer branch mispredictions than unoptimized code. Both these facts lead to higher ROB occupancies. The lower number of overall branch instructions, allow the processor to issue and execute long basic blocks. Since every instruction occupies an entry in the ROB, the more efficient utilization of the pro-

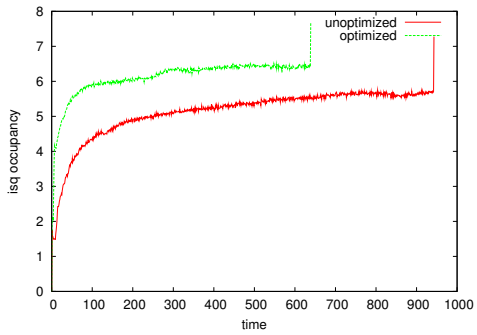

Figure 6. Average ISQ occupancy for *mcf*

cessor resources, leads to higher ROB average occupancies. The decreased number of branch mispredictions also lead to fewer pipeline flushes. Pipeline flushes, albeit detrimental for the application performance, can actually decrease the expected ROB failures at the instance of a flush. During a flush the ROB empties and at that point its instantaneous EF is 0. The lack of flushes in the optimized code results in increased ROB occupancy and thus ROB operates at a higher vulnerability level.

The compounded effect of the increased cache miss rate and decreased branch mispredictions result in higher instantaneous ROB occupancies. As we have already described, this is evident in Figure 5(b), where the ROB occupancy is higher for the optimized code during the entire execution time and the EF ROB values are consistently higher (Figure 5(a)). However, the much shorter execution time for the optimized code compensates for the increased instantaneous vulnerability to soft errors and the average number of expected failures during execution is lower for the optimized application.

bzip2 is the only benchmark which does not comply to our observations, since the optimized version of its code has an increased number of committed and mispredicted branches. The *bzip2* compression algorithm operates on blocks of data in passes and thus, many dependencies exist among its iterations. Therefore, it cannot profit from control-flow optimizations, which instead of enhancing its performance actually harm its branch behavior.

6 Hypothetical Use Case

Our results till now show several distinct trends. First, an optimized application has fewer total expected failures during its execution compared to unoptimized code. However, unoptimized code has lower failure rate in any given time window. Since unoptimized code executes for longer time its overall EF metric is worse. Second optimized code has much higher variance in the expected failures, which potentially forces designers to conservatively allocate more resources to guard against soft errors. In this section we conducted a hypothetical study to evaluate how to exploit lower instantaneous failure rate and lower variance of unoptimized code to improve overall application reliability. For this study we assume that our system has the same FIT rate targets as IBM servers. IBM has set their target Silent Data Corruption (SDC) FIT rate to be 1 failure in 1,000 years or 117 Failures in a billion hours (FITs) and their target Detectable but Unrecoverable Error (DUE) FIT rate to be 1 failure in 25 years, a number that is translated as 4,680 FITs. These numbers reflect the relative importance of both types of errors. Error detection and correction can transform large numbers of SDCs into DUEs. As we can see in Figure 1, the expected number of failures during the benchmarks' execution often surpass the target FIT rates for both SDC and DUE. In this scenario a chip designer may be forced to account for the worst case FIT rate observed during benchmark runs. This accounting may be done by employing more error detection and correction hardware, which may lead to increased hardware costs for providing reliability.

We have noted that the optimized versions of the code have instantaneous failure rates that far exceed the IBM target FIT rates. On the other hand, the unoptimized version of the code has lower instantaneous failure rates and also less variance. Hence, instead of relying on expensive hardware error detection and correction mechanisms we assume a hypothetical machine that selects between optimized and unoptimized versions of the code. Selecting between the two versions of the code at any arbitrary instruction is difficult. There may not be any one-to-one mapping of every instruction in optimized code with an instruction in the unoptimized code. However, there are clearly demarcated regions when it is possible to identify such one-to-one mapping between the two code versions. For instance, when a function call is being made or when a system service is being request it may be possible to switch between the two versions of the code. Assuming that our hypothetical machine can toggle back and forth between the two code versions, we can select an unoptimized version of code for a short window of time when the instantaneous failure rate of the optimized code exceeds a FIT target. By applying this sort of "capping" technique, we can ensure that the application executes within the FIT rate target while still using less error detection and protection hardware than is necessitated by the worst case behavior. Such a dynamic toggle from optimized to unoptimized code comes at the expense of increased execution time.

Figure 8(a) demonstrates this technique on the *parser* benchmark, by setting the cut-off number of expected failures to the SDC FIT target rate (114 FITs). The given FIT rate corresponds to the alloted number of failures for the entire processor. For the given example though, let us assume that the entire quota of failures is given to the ROB, which is the only structure we are applying our technique on. As we

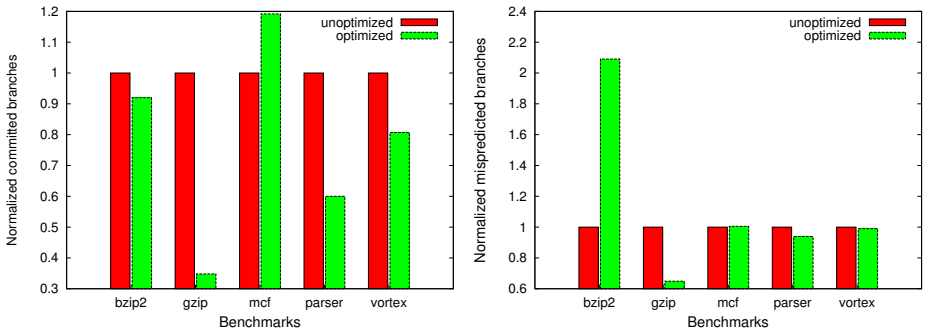

Figure 7. (a) Committed branches (b) Mispredicted branches

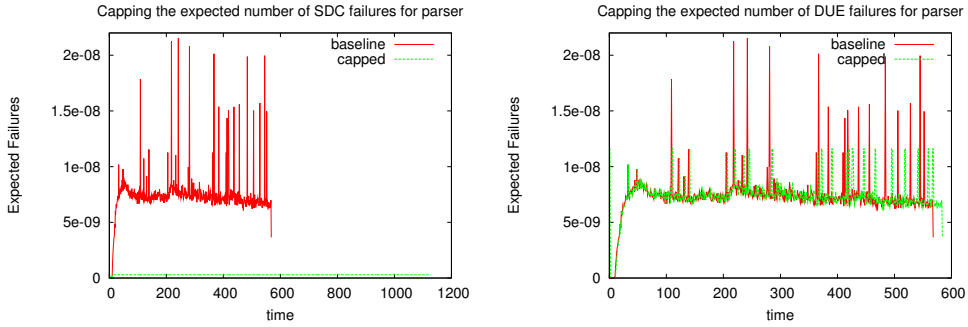

Figure 8. (a) Capping SDC failures for *parser* (b)Capping DUE failures for *parser*

can see, the given cut-off value is consistently lower than the instantaneous EF value of the benchmark across execution and as a result, the system would operate in its unoptimized state for all execution. This would lead to an execution that would be $2\times$ slower, but the EF value would be under the specified FIT rate.

The same approach could be followed by imposing a cut-off for the DUEs as can be seen in Figure 8(b). The higher cut-off value in this case results in less capping and therefore lower time overhead. More specifically, the application executes for only 8.5% at its unoptimized state, which would impose a minimal performance overhead of less than 10%. Moreover, the lower variation and increased predictability are highly desirable properties. Engineers and system developers can apply more easily optimization and reliability techniques on those types of systems.

7 Conclusions/Future Work

In this paper we analyze the relationship between the optimizations applied by the compiler and the effects produced on the system's reliability. We executed various versions of the SPEC benchmarks compiled with different optimization flags of the *gcc* compiler and we looked into the micro-architectural responses that resulted. We observed that the optimized code is usually more compact and executes less redundant instructions leading to better hardware utilization and more instructions in-flight. More efficient utilization in conjunction with significantly lower execution time leads to fewer expected failures during execution. However, the number of expected failures during execution varies across execution time presenting a variable and unpredictable behavior. The unpredictability in the system's behavior obliges designers and architects to plan for the worst case leading to overly restricted and overprovisioned systems.

Armed with the insight of the correlation between compiler optimizations and the processor's reliability, we can tamper the variable time behavior of the system's reliability and strive for a smoother and more predictable behavior. This goal can be achieved either by configuring the hardware or the software running on the system in order to strike a balance between the performance and reliability goals set by the system designers, manufacturers and the users.

Our methodology can be adopted by hardware designers and compiler developers to make it possible to analyze the relationship between the reliability hardware features and compiler optimization. Such an awareness can lead to more reliable hardware or more reliable compiler-optimized code, in addition to the ability to dynamically adjust code reliability for a fixed architecture and compiler. Additionally, we can ensure that the system operates within its Silent Data Corruption (SDC) and Detectable and Unrecoverable Errors (DUE) limits without heavy overheads on performance or hardware complexity.

References

[1] SoftArch: An architecture level tool for modeling and analyzing soft errors. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 496–505, 2005.

[2] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers*, 22(3):258–266, may-june 2005.

[3] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *ISCA '05: Proceedings of the 2005 International Symposium on Computer Architecture*, pages 532–543, 2005.

[4] J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *DSN '08: Proceedings of the 2008 International Conference on Dependable Systems and Networks*, pages 482–491, 2008.

[5] L. Duan, B. Li, and L. Peng. Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics. In *HPCA '09: Proceedings of the 2009 International Symposium on High Performance Computer Architecture*, pages 129–140, 2009.

[6] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 2002 International Symposium on Computer Architecture*, pages 148–157, 2002.

[7] Free Software Foundation. GCC, the GNU compiler collection.

[8] T. Funaki and T. Sato. Formulating MITFfor a multicore processor with SEU tolerance. In *DSD '08: Proceedings of the 2008 Conference on Digital System Design Architectures, Methods and Tools*, pages 234–241, 2008.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '04: Proceedings of the 2004 International Workshop on Workload Characterization*, pages 3–14, 2001.

[10] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, pages 28–35, July 2000.

[11] T. Jones, M. O'Boyle, and O. Ergin. Evaluating the effects of compiler optimisations on AVF. In *INTERACT '08: Proceedings of the 2008 Annual Workshop on the Interaction between Compilers and Computer Architecture in conjunction with HPCA-14*, pages 112–118, 2008.

[12] B. Li, L. Duan, and L. Peng. Efficient microarchitectural vulnerabilities prediction using boosted regression trees and patient rule inductions. *IEEE Transactions on Computers*, pages 593–607, may 2010.

[13] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *ISCA '08: Proceedings of the 2008 International Symposium on Computer Architecture*, pages 341–352, 2008.

[14] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *ISPASS 09: Proceedings of the 2009 International Symposium on Performance Analysis of Systems and Software*, pages 53–64, 2009.

[15] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., 2008.

[16] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the Architectural Vulnerability Factors for a high-performance microprocessor. In *MICRO '03: Proceedings of the 2003 International Symposium on Microarchitecture*, pages 29–42, 2003.

[17] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the 2006 International Symposium on Code Generation and Optimization*, pages 319–330, 2006.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *Transactions on Architecture and Code Optimization*, 2, 2005.

[19] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *ISCA '07: Proceedings of the 2007 International Symposium on Computer Architecture*, pages 516–527, 2007.

[20] J. Yan and W. Zhang. Evaluating instruction cache vulnerability to transient errors. In *MEDEA '06: Proceedings of the 2006 workshop on Memory Performance*, pages 21–28, 2006.

[21] W. Zhang. Computing cache vulnerability to transient errors and its implication. In *DFT '05: Proceedings of the 2005 International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 427–435, 2005.