Evaluating the Effects of Compiler Optimisations on AVF

Timothy M. Jones, Michael F.P. O'Boyle
Member of HiPEAC,
School of Informatics
University of Edinburgh, UK
{tjones1,mob}@inf.ed.ac.uk

Oğuz Ergin
Department of Computer Engineering,
TOBB University of Economics
and Technology, Ankara, Turkey
oergin@etu.edu.tr

Abstract

Transient faults are becoming more of a problem to processor designers as feature sizes shrink and the number of transistors on a chip increases. Significant research has focused on hardware techniques to evaluate and reduce the architectural vulnerability to soft errors (AVF). This paper, however, considers the problem from a different angle, evaluating the effects of compiler optimisations on the AVF of an entire embedded processor. We consider the impact on performance and AVF and produce a new metric (ADS) to evaluate the trade-offs between reducing susceptibility to transient faults and decreasing processor performance. We show that optimisations enabled by default at -O2 and -O3 can lead to large performance decreases, a higher AVF value and an ADS value of over 1.2. However, selectively picking the combination of optimisations means that performance increases can be achieved with negligible effect on AVF, leading to an ADS value of 0.91, with the best combination reducing one benchmark's AVF by 13%.

1 Introduction

As feature sizes shrink and the number of transistors on a chip increases, processors are becoming less reliable and more susceptible to soft errors [1, 2]. These are faults that occur seemingly at random and can be caused by a variety of different events, from strikes by alpha particles in the packaging materials and neutrons from cosmic rays, to power supply noise and leakage effects. These errors do not cause permanent damage to the chip, and as such are often termed transient faults, but can result in a build-up of charge that eventually will alter a signal transition or stored value and lead to incorrect program execution.

Recently, Mukherjee *et al.* [3] proposed a scheme to determine the architectural vulnerability factor (AVF) of a processor structure. The methodology was also extended in [4, 5]. The AVF is defined as the probability that a fault within the structure will result in a visible system error. It is estimated by tracking the structure's bits that are needed for architecturally correct execution (ACE), meaning that an error in an ACE bit would lead to an error in the final program output. By conservatively considering all bits to be ACE unless proved otherwise, an upper bound on AVF or susceptibility to soft errors can be determined.

There have been many architectural schemes to detect and reduce AVF [6, 7, 5]. However, there has been little work in evaluating the effects of the compiler and the optimisations it performs. We address this lack of knowledge by considering the impact of compiler optimisations on the AVF of an entire embedded processor to discover whether the compiler can be of assistance in reducing vulnerability without sacrificing performance.

Although we can evaluate the effects of optimisations on performance and AVF separately, what we really need is a metric that captures the trade-offs between the two. We address this deficiency by proposing a new metric, the AVF-delay-squared product (ADS) which effectively shows the relationship between performance and AVF.

The rest of this paper is structured as follows. Section 2 discusses related work then section 3 describes the processor structures vulnerable to AVF. Section 4 considers the methodology and introduces our new metric, then section 5 presents our results. Finally, section 6 concludes this paper.

2 Related Work

Transient faults are an important problem in microprocessor design and the soft error rate is expected to increase in processors manufactured with future technologies [2]. Prior work has quantified the effect of soft errors on the processor pipeline both at the architectural level [8] and gate levels [9] in this developing area of research.

There are many techniques proposed to reduce soft error vulnerability. Parity and error correcting codes (ECC) are employed to detect and correct single bit upsets in stored data [6], but simply identifying consecutive zeros and ones in generated values can avoid many soft errors [7]. Trying to identify an error by observing the general symptoms of soft errors in the processors was proposed in [10].

Redundancy in space (taking multiple copies of the same data) and redundancy in time (executing the same instruction multiple times) are also used to detect and correct soft errors [5]. Different techniques have been proposed to accomplish redundancy in space [11, 12, 13, 14] and redundancy in time [15, 16], with redundant multithreading a specific example of such schemes [17, 18].

Many compiler optimisations have been proposed and implemented to improve program performance but little work has been focused on soft error vulnerability. One exception was a study into the interaction between optimisations in terms of power and soft error vulnerability [19]. Recently Reis *et al.* proposed using hardware-software hybrid schemes which achieve fault tolerance by replicating instructions at the compiler level and using hardware fault detectors that make use of this redundancy [20, 21].

3 Vulnerable Structures

This section considers the processor structures that contribute the most to the AVF value of the XScale processor [22] that we study and discusses optimisations that will be beneficial to them.

The most significant sources of vulnerability to soft errors are the write, fill and pend buffers that are used when communicating with external memory. Instructions and data are stored in these buffers until the memory bus becomes available. Hence, optimisations that reduce the number of loads and stores will be beneficial for both performance and the AVF of these buffers. This is because the buffers will be used less often and, when they are used, the memory accesses will complete much faster.

Another structure susceptible to soft errors is the register file. Optimisations that reduce register define-use distances will be beneficial for AVF by limiting the time that register values are needed, since registers are no longer vulnerable after their last consumer has read them.

Finally the latches between the pipeline stages are also vulnerable to soft errors. Nops, dead or predicated-false instructions mean a lower AVF because then the latches do not hold valid data [3]. One way to achieve this is to add redundant code to the programs, but this would reduce performance. We seek to address this in section 4 by proposing a metric that trades-off performance and AVF. However, in general, optimisations that reduce the number of pipeline stalls will be beneficial to the AVF of the latches as valid data is held for fewer cycles.

Parameter	Configuration
Pipeline	7/8 Stages
Functional Units	1 ALU, 1 MAC, 1 Load/Store
Issue	Single Issue, In-Order
Commit	Out-of-Order (Scoreboard)
Memory Bus Width	32 Bit
Memory Latency	50 Cycles
I-TLB, D-TLB	32-Entry Fully Associative
I-Cache, D-Cache	32KB, 32-Way, 32B Block
Data Buffers	32B Fill Buffer (Read) and 16B Write Buffer

Table 1. Processor configuration.

4 Methodology

For this study we chose to use gcc [23] as our compiler which has a wide range of optimisation flags. We ran our experiments on the XTREM simulator [24] which has been validated against the Intel XScale processor [22]. Table 1 summarises the main processor parameters. We modified the simulator to enable the calculation of AVF in the same way as in [3]. We considered all benchmarks from the MiBench benchmark suite [25] apart from *ghostscript* which would not run correctly on our system. We ran all benchmarks to completion with the *large* input set.

We used the number of cycles as the performance metric because this is independent of the processor speed. To calculate AVF we used the method proposed by Mukherjee *et al.* [3] and considered the AVF of the whole processor core. We assumed the caches contained error-correction hardware so did not consider them in our AVF calculations.

We also introduce a new metric to evaluate the trade-offs between performance and AVF. We call it the AVF-delay-squared product (ADS) and it is computed by multiplying the change in AVF from the baseline with the square of the change in performance from the baseline.

$$ADS = \frac{Opt\_AVF}{Base\_AVF} \cdot \left(\frac{Opt\_cycles}{Base\_cycles}\right)^2$$

So, if the effect of an optimisation is to increase per-

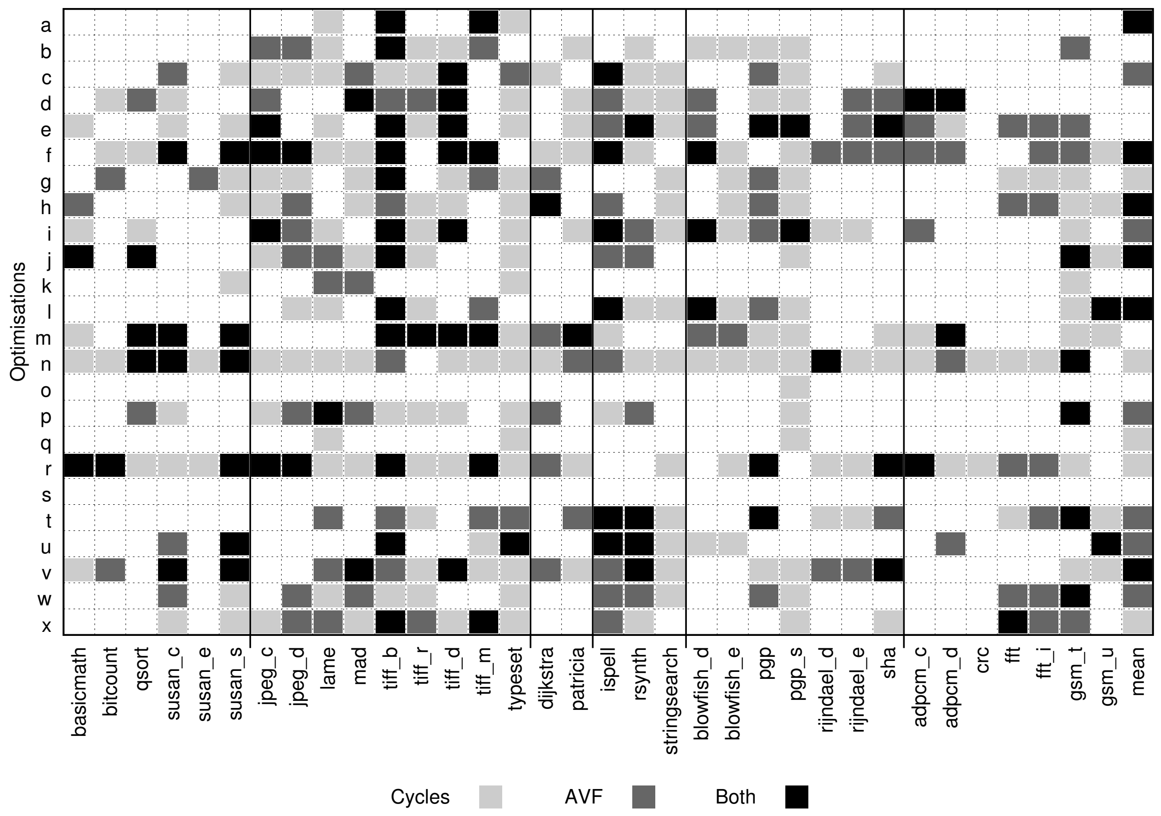

Figure 1. Effects on performance and AVF of the optimisations described in table 2 when added to optimisation -O1. A light grey box indicates that the optimisation in that column is beneficial for performance (cycles) for the benchmark in that row. A medium grey box indicates the same for AVF and a black box shows an optimisation that is beneficial for both. All optimisations are compared with the baseline (-O1).

formance (decrease the number of cycles) then the ADS product will be lower. Likewise if the optimisation decreases AVF. Hence an ADS value of less than one is desirable. When there is a trade-off between performance and AVF then performance increases must take priority since AVF is a secondary concern to processor designers. We want our metric to reflect this and, by squaring the delay, we achieve this aim, favouring optimisations that increase performance over those that decrease AVF.

5 Results

This section presents the results of our experiments. First we consider the effects of the optimisation flags individually in section 5.1. Then we look at the effects of combining flags in section 5.2.

5.1 Individual Optimisations

This section considers the individual optimisation flags that gcc enables to get from *-O1* to *-O3*. We consider 24 optimisations in isolation to understand their effects on performance and AVF. We show the effects of each optimisation graphically in figure 1 with the key to each optimisation letter shown in table 2. In these experiments each benchmark was compiled with *-O1 -fflag*, so for example, for optimisation *f* each benchmark was compiled with *-O1 -fgcse*.

In figure 1, the benchmarks are shown in columns and the different optimisations shown in rows. A light grey box means that the optimisation in that row is beneficial for the benchmark in that column in terms of performance. A medium grey box indicates the same for AVF. Where there is a black box then the optimisation flag is beneficial to both performance and AVF for that benchmark. Also included, in the right-hand

	Optimisation		Optimisation
a	thread-jumps	m	schedule-insns
b	crossjumping	n	schedule-insns2
c	optimize-sibling-calls	o	regmove
d	cse-follow-jumps	p	strict-aliasing
e	cse-skip-blocks	q	delete-null-pointer-checks
f	gcse	r	reorder-blocks
g	expensive-optimizations	s	reorder-functions
h	strength-reduce	t	unit-at-a-time
i	rerun-cse-after-loop	u	tree-pre
j	rerun-loop-opt	v	inline-functions
k	caller-saves	w	unswitch-loops
l	peephole2	x	gcse-after-reload

Table 2. Key to optimisation flags.

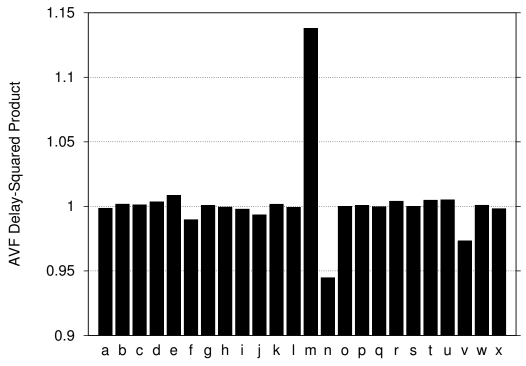

Figure 2. The AVF-delay-squared product for each optimisation added to -O1 across the whole benchmark suite.

column, is the average effect of each optimisation over the whole benchmark suite, calculated using the geometric mean, which can be used as a classification of how each optimisation affects the benchmark suite.

Figure 1 shows that the optimisation flags affect each benchmark differently and are therefore beneficial to some programs and not to others. The optimisations must be picked carefully otherwise they could harm performance and AVF. However, we can consider the programs in their sub-benchmark suites to determine the beneficial optimisations on a coarse-grained basis.

Security For the *security* suite (*blowfish*, *pgp*, *rijndael* and *sha*) common sub-expression elimination optimisations (d, e and f) are beneficial. These reduce duplicated control flow and register define-use distances, decreasing the AVF of the register file. For example, optimisation f on *blowfish_d* reduces the register file AVF by 5%.

Office The three benchmarks making up the *office* suite (*ispell*, *rsynth* and *stringsearch*) are interesting. For the first two, common sub-expression elimination is important again, but so are t, u, v and w. In fact, optimisation u, which performs partial redundancy elimination on trees, reduces the AVF of the buffers by 8% in *ispell* and 25% in *rsynth*. Inlining functions (v) is also an important optimisation for this benchmark, reducing the register file AVF by 5%. This is because the call and return overheads are eliminated, meaning more efficient register usage. However, for *stringsearch*, none of the optimisations are beneficial for AVF. This benchmark uses tables to search for strings and the optimisations that improve performance actually increase the number of loads and stores, raising the AVF of the fill and write buffers.

Consumer The *consumer* benchmarks (*jpeg*, *lame*, *mad*, *tiff* and *typeset*) provide the most scope for optimisation. These are the most complex benchmarks and most optimisations have some sort of positive effect on performance or AVF. It is interesting to discuss the *tiff* group of benchmarks which all share the same library but are affected differently by the optimisations. This is because of the algorithms used within each program. In *tiff_b* and *tiff_d* there is one large nested loop which dominates execution. Again, common sub-expression elimination is beneficial, this time increasing the number of dynamically dead instructions (by 22% for optimisation e on *tiff_b*) and therefore reducing the AVF of the latches. In *tiff_r* and *tiff_m*, however, there is more linear control flow and optimisations have less impact.

Network For the *network* benchmarks (*dijkstra* and *patricia*), scheduling instructions (m) is beneficial for AVF. For these benchmarks register usage is improved by attempting to eliminate stalls due to data unavailability. For *dijkstra*, this decreases the register file AVF by 5%. However, surprisingly, the related optimisation n, which performs another pass after register allocation, fails to alter register file AVF in either benchmark.

Impact on ADS It is also interesting to consider the effects of each optimisation on the ADS metric. Figure 2 shows the results for each optimisation with the geometric mean taken across the whole benchmark suite. It is clear to see that the majority of optimisations have little effect on ADS, although two in particular stand out. Optimisation m produces a poor ADS value of 1.14. Although, as we have seen, this optimisation works well for the *network* benchmarks, it tends to increase the number of stores executed, increasing the AVF of the buffers. It seems that for the *network* benchmarks, optimisation m is better than n, but for all others it is the opposite way around.

Summary Having considered the impact of opti-

Metric	Optimisations
Cycles	a f g h j l n q r v x
AVF	a c f h i j l p t u v w
Both cycles & AVF	a f h j l v

Table 3. The optimisations that, in general, improve cycles, AVF or both.

misations on the sub-benchmark suites we can classify the compiler flags as being generally good for cycles, AVF or both. Table 3 shows the classifications.

In general, the optimisations performing common sub-expression elimination and variants (f, i and u) are beneficial for AVF. In addition, optimisation f improves cycles and is beneficial for both performance and AVF. Optimisation v is also beneficial for both. This turns on function inlining which reduces the function call overhead, improving register usage and reducing the AVF of the register file. It also reduces the number of loads and stores required (for saving registers across the function calls), which improves performance and the AVF of the fill and write buffers.

5.2 Combining Flags

Having considered the effects of the individual optimisation flags on performance, AVF and ADS product, we now combine the flags to discover the interaction of the optimisations with each other. We chose three combinations of flags on a per benchmark basis, guided by the optimisation effects shown in figure 1. Firstly, those that decrease the number of cycles needed (increase performance); secondly those that decrease AVF; and thirdly those that are beneficial for both. Figure 3 shows the impact on cycles, AVF and ADS product. For comparison we also show results from default optimisation combinations *-O2* and *-O3*.

Default optimisations It is clear from this graph that, surprisingly, the default optimisation levels perform badly, increasing both cycles and AVF, and leading to an ADS product of over 1.2. In fact, since we have normalised to the baseline of *-O1* it is actually better for performance and AVF to apply no optimisations to these programs. In terms of AVF, these optimisations are detrimental because they increase the number of loads and stores, leading to increases in the AVF of the write and fill buffers. For example, *rijndael_e* has 130% more loads with *-O2*, raising the AVF of the buffers from 0.05 to 0.40.

Best for cycles Being more selective with the optimisation flags that are enabled can, however, bring benefits. Selecting the optimisations on a per-program

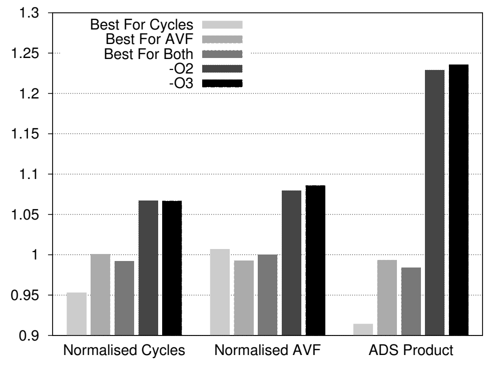

Figure 3. Normalised cycles, AVF and AVF-delay-squared product for each optimisation across the whole benchmark suite for optimisations that improve cycles, AVF or both.

basis that are best for cycles gives nearly a 5% performance increase. This has a negligible AVF increase over the baseline and achieves a good ADS value of 0.91 on average.

Best for AVF and ADS The results for the optimisations that are best for AVF and those that are best for both performance and AVF are disappointing. Although there are no performance decreases, AVF is only marginally reduced in these cases. It seems that the register file AVF is reduced for the majority of benchmarks because the optimisations improve register usage, reducing the define-use distances. For one benchmark, *patricia*, the overall AVF is reduced by 13% with the AVF of the latches reduced by 19% as the optimisations increase speculation and the number of predicated-false instructions by 29%. However, the AVF of other benchmarks increases. For example, for *fft* it increases by 5%, due to the interactions between the optimisation flags. Although optimisation e is beneficial for AVF by removing duplicated control flow, optimisations r and w move code around and duplicate it where needed, increasing the number of loads executed and the AVF of the fill and write buffers. Overall these combinations of optimisations achieve, on average, ADS products of just 0.99 and 0.98.

6 Conclusions

This paper has evaluated the impact of compiler optimisations on performance and issue queue AVF. We have shown that optimisation flags considered individually are difficult to classify, since they affect benchmarks in different ways. Using our new metric, AVF-delay-squared product (ADS), we show that most optimisations have little impact over the whole benchmark suite, although one is particularly detrimental and two are, in general, beneficial.

Having used a coarse-grained approach to classification of the individual optimisation flags, We have combined those that are beneficial for performance, AVF and both on a per-program basis, finding that, in general, performance can be improved but there is little impact on AVF. We compared these combinations with default optimisation levels of *-O2* and *-O3* and discovered that the latter are detrimental for performance and AVF, increasing the number of loads executed and, hence, the AVF of the fill and write buffers. These default optimisations give ADS products of 1.2 over the whole benchmark suite. Our combinations, however, give an ADS value of 0.91, with the best benchmark achieving a 13% decrease in AVF for one combination of flags.

For future work we plan to investigate the interactions between these flags, enabling us to draw further detailed conclusions about the effects of the optimisations on the different types of program being optimised.

References

[1] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of DSN*, 2002.

[2] R. Baumann. Soft errors in advanced computer systems. In *IEEE Design and Test of Computers*, 2005.

[3] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of MICRO-36*, 2003.

[4] A. Briwas et al. Computing architectural vulnerability factors for address-based structures. In *ISCA*, 2005.

[5] H. Asadi et al. Balancing performance and reliability in the cache hierarchy. In *ISPASS*, 2005.

[6] R. Phelan. Addressing soft errors in ARM core-based designs. Technical report, ARM, 2003. White Paper.

[7] O. Ergin et al. Exploiting narrow values for soft error tolerance. *Computer Architecture Letters*, 5, 2006.

[8] N. Wang et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN*, 2004.

[9] K. Constantinides et al. Assessing SEU vulnerability via circuit-level timing analysis. In *Workshop on Architectural Reliability*, 2005.

[10] N. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *DSN*, 2005.

[11] G Memik et al. Increasing register file immunity to transient errors. In *DATE*, 2005.

[12] W. Zhang et al. ICR: In-cache replication for enhancing data cache reliability. In *DSN*, 2003.

[13] J. Hu et al. In-register duplication: Exploiting narrow-width value for improving register file reliability. In *DSN*, 2006.

[14] M. Gomaa et al. Transient-fault recovery for chip multiprocessors. In *ISCA*, 2003.

[15] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *MICRO*, 1999.

[16] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.

[17] A. Gomaa, M and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA*, 2005.

[18] M. K. Qureshi et al. Microarchitecture-based introspection: A technique for transient fault tolerance in microprocessors. In *DSN*, 2005.

[19] L. Li et al. Soft error and energy consumption interactions: A data cache perspective. In *ISLPED*, 2004.

[20] G. A. Reis et al. Swift: Software implemented fault tolerance. In *CGO*, 2005.

[21] G. A. Reis et al. Design and evaluation of hybrid fault-detection systems. In *ISCA*, 2005.

[22] Intel Corporation. Intel XScale microarchitecture. http://www.intel.com/design/intelxscale/.

[23] Free Software Foundation. GCC, the GNU compiler collection. http://gcc.gnu.org.

[24] G. Contreras et al. XTREM: a power simulator for the Intel XScale core. In *LCTES*, 2004.

[25] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4 (MICRO-34)*, 2001.