Universitat Autònoma de Barcelona

Departament d'Arquitectura de Computadors i Sistemes Operatius

Màster en
Ciència i Enginyeria Computacional

Mejorando la robustez de aplicaciones frente a fallos transitorios

Memoria del trabajo de investigación del "Máster en Ciencia y Ingeniería Computacional", realizada por João Gramacho, bajo la dirección de Dolores Rexachs presentada en la Escuela de Ingeniería (Departamento de Arquitectura de Computadores y Sistemas Operativos)

2010

Iniciación a la investigación. Trabajo de fin de máster

Máster en Ciencia e Ingeniería Computacional

Mejorando la robustez de aplicaciones frente a fallos transitorios

Realizada por João Gramacho en la Escuela de Ingeniería, en el Departamento Arquitectura de Computadores y Sistemas Operativos

Dirigida por: Dolores Rexachs

Firmado

Directora	Estudiante
Dolores Rexachs	**João Gramacho**

Agradecimientos

Agradezco a todos los compañeros de CAOS, en especial a los de mi grupo de trabajo, y, particularmente, a Dolores Rexachs y Emilio Luque por creer en mi trabajo.

Muchas gracias Dona Ana, Sérgio, Gabi, Anamel, Laís y toda mi familia y amigos que, mismo desde tan lejos, están siempre cerca.

Agradezco también a los amigos de Barcelona, mi "nueva familia", con quien tengo disfrutado los tiempos de ocio y vacaciones.

Finalmente, agradezco a mi Graziela por la paciencia, cariño y motivación que ella me dio para estar aquí terminando este trabajo.

Abstract

Computer chips implementation technologies evolving to obtain more performance are increasing the probability of transient faults. As this probability grows and on-chip solutions are expensive or tend to degrade processor performance, the efforts to deal with these transient faults in higher levels (such as the operating system or even at the application level) are increasing. Mostly, these efforts are trying to avoid silent data corruptions using hardware, software and hybrid based techniques to add redundancy to detect the errors generated by the transient faults. This work presents our proposal to improve the robustness of applications with source code based transformation adding redundancy. Also, our proposal takes account of the tradeoff between the improved robustness and the overhead generated by the added redundancy.

Key words: transient faults, fault detection, robustness, source code transformation.

Resum

L'evolució de processadors a la recerca d'un major rendiment està augmentant la probabilitat de fallades transitòries. Al costat del creixement d'aquesta probabilitat, els esforços per fer front a tals fallades en tots els nivells (com el sistema operatiu o fins i tot a nivell d'aplicació) també estan en augment. Els esmentats esforços estan tractant d'evitar, sobretot, la corrupció silenciosa de dades. En aquest treball presentem la nostra proposta per a millorar la robustesa d'aplicacions basant-nos en canvis en el codi font de l'aplicació, inserint redundància de còmput, i tenint en compte la relació que existeix entre la redundància afegida i la sobrecàrrega generada en el temps d'execució de l'aplicació.

Paraules clau: fallades transitòries, detecció de fallades, robustesa, transformació del codi font.

Resumen

La evolución de procesadores en busca de un mayor rendimiento está aumentando la probabilidad de fallos transitorios. Junto al crecimiento de esta probabilidad, los esfuerzos para hacer frente a tales fallos en todos los niveles (como el sistema operativo o incluso a nivel de aplicación) también están en aumento. Dichos esfuerzos están tratando de evitar, sobre todo, la corrupción silenciosa de datos. En este trabajo presentamos nuestra propuesta para mejorar la robustez de aplicaciones basándonos en cambios en el código fuente de la aplicación, insertando redundancia de cómputo, y teniendo en cuenta la relación que existe entre la redundancia añadida y la sobrecarga generada en el tiempo de ejecución de la aplicación.

Palabras clave: fallos transitorios, detección de fallos, robustez, transformación de código fuente.

Contenido

Capítulo 1 Introducción 1
1.1 Introducción 1
1.2 Objetivo 2
1.3 Organización de este documento 3
Capítulo 2 Fallos transitorios 5
2.1 El concepto de fallos transitorios 5
2.2 Los efectos de un fallo 6
2.3 Los efectos de los fallos transitorios 7
2.3.1 Excepción de instrucción inválida/Cambio en una instrucción 9
2.3.2 Error de paridad durante la lectura 10
2.3.3 Violación de acceso a la memoria 11
2.3.4 Cambio de instrucciones, datos y direcciones de memoria 11
2.4 Robustez y prestaciones 11
Capítulo 3 Caracterización de la robustez 13
3.1 Introducción 13
3.2 Usando inyección de fallos para caracterizar la robustez 14
Capítulo 4 Estado del arte 19
4.1 Introducción 19
4.2 Detección de fallos transitorios 19
4.2.1 Detección basada en hardware 20
4.2.2 Detección basada en software 23
4.3 Donde y como hemos elegido añadir la redundancia 29
Capítulo 5 Mejorando la robustez 31
5.1 Introducción 31
5.2 El caso base: la regla DIC 31
5.3 Nuevas reglas de transformación propuestas 33
5.3.1 Regla V 33
5.3.2 Reglas EAD/EDI 34
5.3.3 Reglas IDD/IDI 36
5.3.4 Reglas TOD/TOI 38
5.3.5 Regla TOH 40
5.4 Automatizando la transformación del código fuente 41
Capítulo 6 Evaluación experimental 43
6.1 Introducción 43

6.2 Eligiendo que transformar 44
6.2.1 Transformación de la aplicación CG 45
6.2.2 Transformación de la aplicación FT 45
6.2.3 Transformación de la aplicación SP 45
6.3 Análisis general de los resultados 45
6.3.1 Experimentación con la optimización más agresiva (O3) 46
6.3.2 Experimentación con optimización fuerte (O2) 50
6.3.3 Experimentación con optimización moderada (O1) 55
6.3.4 Análisis general de la robustez 60
6.3.5 Análisis general del MWTF 62
6.4 Comparación con trabajos del estado del arte 64
Capítulo 7 Conclusión y trabajos futuros 67
7.1 Conclusión 67
7.2 Trabajos futuros 68
Referencias 69

Lista de figuras

Figura 1 – Fallos, errores y averías. 6
Figura 2 – Clasificación de lo que puede pasar frente a un fallo transitorio. 7
Figura 3 – Posibles efectos de un *soft error*. 9
Figura 4 – Fase de preparación de la caracterización de la robustez. 14
Figura 5 – Ejecuciones con inyección de fallos. 15
Figura 6 – Niveles donde se puede estar el mecanismo de detección de fallos. 20
Figura 7 – Redundancia modular con dos y tres procesadores. 21
Figura 8 – Categorización de donde y cuando si añade la redundancia en las aplicaciones. 24
Figura 9 – Sobrecarga, mejoría en la robustez y MWTF obtenido con el EDDI. 26
Figura 10 – Sobrecarga, mejoría en la robustez y MWTF de SWIFT y EDDI mejorado. 27
Figura 11 – Sobrecarga, mejoría en la robustez y MWTF de ESoftCheck y FullRep (SWIFT). 27
Figura 12 – Diagrama del *framework* Spot. 28
Figura 13 – Arquitectura de los procesadores de los supercomputadores de la lista Top 500. ... 30
Figura 14 – Código fuente ejemplo sin detección de fallos. 31
Figura 15 – Código fuente ejemplo con la regla DIC. 32
Figura 16 – Código fuente ejemplo con la regla DICV. 34
Figura 17 - Código fuente ejemplo con la regla EAD. 35
Figura 18 – Código fuente ejemplo con la regla EAI. 35
Figura 19 – Código fuente ejemplo con la regla IDD. 37
Figura 20 – Código fuente ejemplo con la regla IDI. 37
Figura 21 – Código fuente ejemplo con la regla TOD. 39
Figura 22 – Código fuente ejemplo con la regla TOI. 39
Figura 23 – Código fuente ejemplo con la regla TOH. 41
Figura 24 – Diagrama de la automatización del cambio del código fuente. 42
Figura 25 – Robustez de la aplicación CG con optimización O3. 46
Figura 26 – Tiempo de ejecución normalizado de la aplicación CG con optimización O3. 46
Figura 27 – MWTF normalizado de la aplicación CG con optimización O3. 47
Figura 28 – Robustez de la aplicación FT con optimización O3. 47
Figura 29 – Tiempo de ejecución normalizado de la aplicación FT con optimización O3. 48
Figura 30 – MWTF normalizado de la aplicación FT con optimización O3. 48
Figura 31 – Robustez de la aplicación SP con optimización O3. 49
Figura 32 – Tiempo de ejecución normalizado de la aplicación SP con optimización O3. 49
Figura 33 – MWTF normalizado de la aplicación SP con optimización O3. 50
Figura 34 – Robustez de la aplicación CG con optimización O2. 50
Figura 35 – Tiempo de ejecución normalizado de la aplicación CG con optimización O2. 51

Figura 36 – MWTF normalizado de la aplicación CG con optimización O2. 51
Figura 37 – Robustez de la aplicación FT con optimización O2. 52
Figura 38 – Tiempo de ejecución normalizado de la aplicación FT con optimización O2. 53
Figura 39 – MWTF normalizado de la aplicación FT con optimización O2. 53
Figura 40 – Robustez de la aplicación SP con optimización O2. 54
Figura 41 – Tiempo de ejecución normalizado de la aplicación SP con optimización O2. 54
Figura 42 – MWTF normalizado de la aplicación SP con optimización O2. 55
Figura 43 – Robustez de la aplicación CG con optimización O1. 55
Figura 44 – Tiempo de ejecución normalizado de la aplicación CG con optimización O1. 56
Figura 45 – MWTF normalizado de la aplicación CG con optimización O1. 56
Figura 46 – Robustez de la aplicación FT con optimización O1. 57
Figura 47 – Tiempo de ejecución normalizado de la aplicación FT con optimización O1. 58
Figura 48 – MWTF normalizado de la aplicación FT con optimización O1. 58
Figura 49 – Robustez de la aplicación SP con optimización O1. 59
Figura 50 – Tiempo de ejecución normalizado de la aplicación SP con optimización O1. 59
Figura 51 – MWTF normalizado de la aplicación SP con optimización O1. 60
Figura 52 – Mejoría de la robustez de la aplicación CG. 60
Figura 53 – Mejoría de la robustez de la aplicación FT. 61
Figura 54 – Mejoría de la robustez de la aplicación SP. 61
Figura 55 – Mejoría de la robustez de la aplicación SP transformando varias funciones. 62
Figura 56 – MWTF de los experimentos con la aplicación CG. 63
Figura 57 – MWTF de los experimentos con la aplicación FT. 63
Figura 58 – MWTF de los experimentos con la aplicación SP transformando varias funciones. 63
Figura 59 – Comparación general de las sobrecargas. 64
Figura 60 – Comparación general de la mejoría en la robustez. 65
Figura 61 – Comparación general de los MWTF normalizados. 65
Figura 62 – Comparación general de los %SDC obtenidos. 66

Lista de ecuaciones

Ecuación 1 – Fórmula general del Mean Work to Failure – MWTF. 12
Ecuación 2 – Uso del MWTF en la caracterización de la robustez. 17

Lista de tablas

Tabla 1 – Clasificación de un experimento de inyección de fallos 16
Tabla 2 – Ejemplo de redundancia con EDDI. 25
Tabla 3 – Distribución de las arquitecturas de procesadores de la lista Top 500. 29
Tabla 4 – Versiones de las aplicaciones experimentadas. 43
Tabla 5 – Optimizaciones del compilador utilizadas. 44
Tabla 6 – Funciones transformadas en nuevo experimento con la aplicación SP. 62
Tabla 7 – Dados obtenidos para la comparación de trabajos. 64

Capítulo 1 Introducción

1.1 Introducción

El constante aumento de la densidad de los procesadores de computadores en las últimas décadas es uno de los factores responsables de las mejoras en la capacidad de procesamiento. Los procesadores están usando transistores cada vez más pequeños, más transistores, y también operando con voltaje más bajo. El efecto secundario de este escenario es que los procesadores se vuelven menos robustos contra fallos transitorios [**1**].

Los fallos transitorios son los fallos que pueden ocurrir sólo una vez y no volver a pasar de la misma manera en la vida útil de un sistema. Los fallos transitorios en los computadores pueden ocurrir en los procesadores, los elementos de la jerarquía de memoria, los buses internos y los dispositivos, a menudo ocasionando una inversión de un bit [**2**]. La radiación cósmica, la alta temperatura de operación y las variaciones en el voltaje del procesador son las causas más comunes de fallos transitorios en computadores [**3**].

Un fallo transitorio puede hacer que una aplicación tenga un comportamiento anómalo (por ejemplo, escribir en una posición de memoria no válida o intentar de ejecutar una instrucción inexistente). Estas aplicaciones que se comportan anómalamente mal serán abruptamente interrumpidas por el sistema operativo provocando una parada inesperada pero segura. Sin embargo, el mayor riesgo para las aplicaciones, es que continúe la ejecución y ocurra una corrupción de los datos silenciosa. Esto ocurre cuando los resultados de las aplicaciones generan un resultado incorrecto que no puede ser explicado o peor que puede no ser detectado [**4**].

En computación de altas prestaciones, el riesgo de tener un fallo transitorio crece con la cantidad de procesadores de computadores trabajando juntos [**5**]. Por lo tanto, cuanta más potencia logra un supercomputador añadiendo más procesadores, mayor será el riesgo de que una corrupción de datos producida por un fallo transitorio pase desapercibida [**6**].

La investigación sobre los fallos transitorios se inició con los computadores en ambientes hostiles [**7**], como el espacio, pero a partir del año 2000 se han hecho públicos informes con los efectos de los fallos transitorios en instalaciones de supercomputación. Estos informes evidencian el riesgo de fallos transitorios en la computación de altas prestaciones, debido al gran número de componentes que trabajan en conjunto.

Usualmente, para mejorar la robustez de aplicaciones en presencia de fallos transitorios, se usan técnicas de redundancia. Estas técnicas pueden afectar al hardware, la capa de software o ambas [**8**]. En los trabajos basados en el software, hay la alternativa de trabajar con cambios realizados

añadiendo redundancia en el código binario generado, tanto estáticamente como dinámicamente [**9**].

Otra alternativa es trabajar realizando cambios basándose en el código fuente de la aplicación. En este caso, hay propuestas para realizar cambios a nivel de compilador que veremos en el estado del arte, donde analizaremos las propuestas EDDI [**10**], SWIFT [**11**] y ESoftCheck [**12**], todas para añadir la redundancia durante la compilación del programa, y hay también propuestas para realizar cambios en el código fuente de la aplicación, de las que también analizaremos [**13**] en el estado del arte. La alternativa de realizar cambios en el código fuente es buena para que los desarrolladores puedan elegir si quieren que la aplicación sea más o menos robusta, dependiendo de las reglas de transformación del código fuente propuestas.

En la literatura hay propuestas un conjunto de reglas de transformación que mejora la robustez de las aplicaciones teniendo en cuenta la relación que existe entre la redundancia añadida y la sobrecarga generada en el tiempo de ejecución de la aplicación [**14**]. Pero en algunos casos, las optimizaciones hechas por los compiladores en busca de mejores prestaciones disminuyen la efectividad de la redundancia añadida en el código fuente, cuando no la saca del todo.

Además, las alternativas basadas en redundancia a nivel del uso de los registros del procesador, casi siempre usan procesadores superescalares, que no son usados en la mayoría de los grandes centros de supercomputación [**15**]. Estos procesadores superescalares poseen gran cantidad de registros disponibles para uso general lo que hace que con estas alternativas no comprometan tanto las prestaciones por la sobrecarga añadida con redundancia basada la duplicación del uso de los registros.

1.2 Objetivo

Nuestro objetivo en este trabajo es mejorar la robustez de las aplicaciones frente a la presencia de fallos transitorios a través de nuevas reglas de transformación del código fuente de las aplicaciones, que permitirán a las aplicaciones detectar errores generados por fallos transitorios y gestionará esta detección, en un primer momento haciendo una parada segura de la aplicación, evitando la propagación del error en el resultado final de la ejecución de la aplicación, mejorando también el equilibrio entre la sobrecarga añadida por los mecanismos de detección y la mejoría en la robustez de la aplicación.

1.3 Organización de este documento

Este documento contiene siete capítulos. En el próximo capítulo presentamos una visión general sobre los fallos transitorios y conceptos relacionados con la investigación sobre los fallos transitorios que usaremos a lo largo de este documento.

El Capítulo 3 presenta la metodología utilizada en la caracterización de la robustez de las aplicaciones en presencia de fallos transitorios y también como evaluaremos el equilibrio entre la pérdida de prestaciones añadida por la redundancia en función de la mejoría de la robustez.

En el Capítulo 4 presentamos el estado del arte sobre detección de fallos transitorios, analizando principalmente las alternativas basadas en software.

El Capítulo 5 presenta nuestras propuestas para mejorar la robustez de las aplicaciones en presencia de fallos transitorios.

Presentamos en el Capítulo 6 la evaluación experimental para valorar nuestra propuesta para mejorar la robustez de aplicaciones en presencia de fallos transitorios y comparamos nuestros resultados con algunos de los presentados en el Capítulo 4.

Finalmente, en el Capítulo 7 presentamos nuestras conclusiones y algunas posibilidades de trabajos futuros.

Capítulo 2 Fallos transitorios

2.1 El concepto de fallos transitorios

Fallos transitorios son aquellos que no reflejan un mal funcionamiento permanente. Un fallo permanente en algún componente hará con que se produzcan fallos, errores o comportamientos inesperados todas las veces que se utilice el componente defectuoso. Por otro lado, los fallos transitorios pueden ocurrir una sola vez en toda la vida útil de los componentes y no volver a ocurrir, ya que son el resultado de influencias externas, como partículas de radiación que producen cambios de voltaje en los circuitos digitales, o algunas fuentes internas como la interferencia en el suministro de energía o la variación de la temperatura [**1**].

Los fallos transitorios generados por la influencia de radiación, por ejemplo, se deben a las partículas energéticas (como los neutrones de la atmósfera), generando acumulación de energía a medida que pasan a través de un dispositivo semiconductor. En este caso, es posible que se acumule energía con carga suficiente para invertir el estado de un dispositivo lógico, inyectando un fallo en el funcionamiento del circuito (por ejemplo, invertir un bit en una posición de memoria o en un registro del procesador) [**8**].

Los fallos transitorios empezaron como un problema para los diseñadores de sistemas de alta disponibilidad y sistemas para entornos hostiles como el espacio exterior [**1**], pero esta situación ha cambiado. A medida que el proceso de miniaturización de los componentes sigue evolucionando, la robustez de estos componentes tras fallos transitorios disminuye. Con el aumento de influencia de la radiación muchos sistemas empezaron a aplicar detección de errores de forma extensiva o empezaron a aplicar mecanismos de corrección, sobre todo para los chips de memorias. El principal problema, es que la protección solamente en la memoria no es suficiente para las tecnologías de miniaturización bajo 65nm.

La necesidad de protección contra los efectos fallos transitorios en la computación comercial y aplicaciones de comunicación están motivando nuevos mecanismos en el chip para proteger sus memorias internas. Eventualmente será necesario incluso una cierta protección en la lógica combinatoria de los chips de computadores, ya que cada vez más aumenta la cantidad de transistores usados en las nuevas tecnologías en búsqueda de mejores prestaciones [**16**].

Pero, con la llegada de los chips *multicore* y *manycore*, la cantidad de transistores y puentes en un procesador de un computador es tan grande que la industria de los chips de ordenador espera que las capas superiores de un computador (el sistema operativo, los *frameworks* de

computación paralela e incluso las aplicaciones) se preparen para trabajar con la posibilidad de que fallos transitorios ocurran con más frecuencia.

2.2 Los efectos de un fallo

Un fallo puede generar uno o más errores latentes. Un error es la manifestación de un fallo en un sistema. Un error latente se torna efectivo una vez que el recurso con el error es utilizado por el sistema para hacer algún tipo de cómputo. Además, un error efectivo a menudo se propaga de un componente del sistema a otro, creando así nuevos errores. Una avería es la manifestación de un error en el servicio prestado por el sistema. Una avería se produce cuando el comportamiento de un sistema se desvía del comportamiento especificado en su construcción.

Por ejemplo, de acuerdo con los estados de la Figura 1:

1. Si una partícula energética golpea una celda de memoria DRAM puede producir fallo;
2. Una vez que este fallo cambie el estado de la celda de memoria DRAM se produce un error latente;
3. Este error se mantiene latente hasta que la memoria afectada sea leída por algún proceso, convirtiéndose en un error efectivo;
4. Una avería se produce si la memoria cambiada por el error es leída y afecta el funcionamiento del sistema o de la aplicación, cambiando su comportamiento.

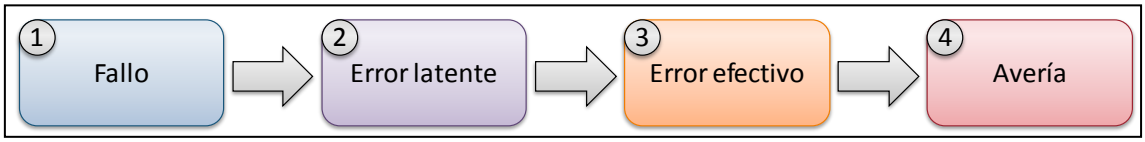

Figura 1 – Fallos, errores y averías.

Los fallos pueden ser caracterizados por su duración como permanentes, transitorios o intermitentes. Un fallo permanente se mantendría en el sistema hasta que sean tomadas medidas correctoras. Fallos intermitentes son los que siguen apareciendo y desapareciendo en algunas circunstancias. Tal como en los fallos permanentes, se puede identificar el aparato defectuoso y tomar medidas correctivas (haciendo una substitución por ejemplo) para impedir que aparecen nuevos fallos intermitentes. Por otro lado, un fallo transitorio aparece, desaparece y probablemente nunca vuelva a ocurrir la misma manera en la vida útil de un sistema [**8**].

Los errores producidos por fallos transitorios se llaman *soft errors*. La observación de un *soft error* no implica que el sistema sea menos fiable que antes. Los *soft errors* pueden cambiar datos pero no pueden cambiar el circuito físico en sí. Si los datos son escritos nuevamente, el circuito seguirá funcionando perfectamente.

La expresión *soft error* utilizada en la literatura de fallos transitorios no debe confundirse con errores de aplicaciones de software (errores de software de programación).

2.3 Los efectos de los fallos transitorios

La Figura 2, adaptada de[**4**], trae una clasificación de lo que puede pasar a un procesador de un computador frente a la ocurrencia de un fallo transitorio generado por una partícula energética.

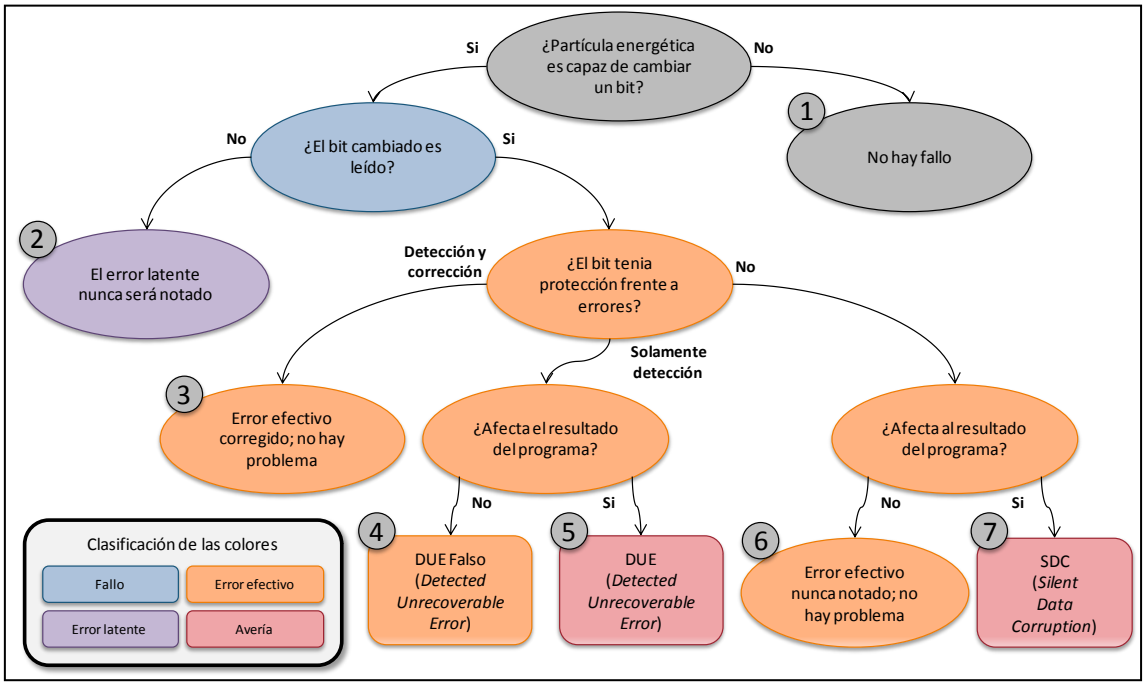

Figura 2 – Clasificación de lo que puede pasar frente a un fallo transitorio.

El ítem 1 de la of Figura 2 indica la posibilidad donde la partícula energética no fue capaz de llegar a cambiar el bit y, por lo tanto, no ha generado ningún fallo.

El error latente pasará cuanto la partícula energética sea capaz de cambiar el estado del bit de la memoria, del registro del procesador o de un circuito interno de cómputo de un procesador.

Si pasa que este bit que ha fallado no es leído por el sistema o es sobrescrito en algún momento después del fallo y antes de su uso, el error latente generado por este fallo nunca será notado (ítem 2 de la Figura 2).

Cuando este bit fallado es leído por el sistema o por alguno de sus componentes, el *soft error* entonces es llamado error efectivo

Un error efectivo, generado por un fallo transitorio, puede pasar sin ser notado por las capas superiores del componente que lo está leyendo si el dicho bit tiene algún mecanismo de detección y corrección (ítem 3 de la Figura 2). Este es el caso, por ejemplo, de las memorias que trabajan con codificación para corrección (ECC). Las memorias de acceso aleatorio dinámico

(DRAM), por ejemplo, es un tipo de dispositivo de memoria que usa ECC para mejorar su fiabilidad por su simplicidad estructural. Usan bits de memoria extra que son usados por los controladores de la memoria para almacenar la paridad de conjuntos de bits.

Si el bit fallado posee solamente detección de errores, el fallo puede generar una condición que llamamos *Detected Unrecoverable Error* (DUE), evitando la generación de resultados corrompidos. En el caso de que se pase un DUE, el sistema ha leído el bit fallado, sabe que este bit tiene un error y que no hay un mecanismo para hacer la corrección de este error. El ítem 4 de la Figura 2 representa la situación donde el *soft error* no afectaría el resultado de la aplicación y por eso es considerado un DUE falso.

Pero, si el *soft error* afectar al resultado computado por el programa ejecutando en el sistema (ítem 5 de la Figura 2), el sistema tendrá que informar sus capas superiores (probablemente al sistema operativo) del error efectivo, evitando así que el programa continúe su ejecución en estas condiciones. Como este error seguramente afectaría el resultado del programa que estaba ejecutando, es llamado simplemente DUE. Cuando un bit que estaba asignado o utilizado por la ejecución de un determinado programa se considera que ha fallado por las capas inferiores, es común que el sistema operativo pare la ejecución de la aplicación (*process kill*) y considere que esta parada fue necesaria por un comportamiento indebido de la aplicación, pero todo el resto del sistema, incluyendo del sistema operativo y otras aplicaciones, pueden seguir ejecutando normalmente.

Entretanto, si el bit fallado asignado alocado para el sistema operativo, puede ser que la parte que no si puede recuperar del sistema solamente permita que el sistema operativo siga ejecutando con normalidad después de que el sistema vuelva a reiniciar por completo (*system kill*), reiniciando el sistema operativo por completo, en este caso todas las aplicaciones que se estaban ejecutando tuvieron que ser interrumpidas.

Pero el fallo que puede traer más problemas, es cuando un sistema es afectado por un fallo y el *soft error* ha pasado en un componente del sistema sin ningún tipo de protección. En este caso, el bit fallado puede acabar por ser usado por una aplicación ejecutando en el sistema.

El ítem 6 de la Figura 2 representa la situación en la que un *soft error* no detectado por el sistema no afecta el resultado generado por el programa en ejecución.

Si el sistema usa el bit fallado en su operación sin saber que este bit ha sido cambiado, el sistema estará pasando por una corrupción de dados silenciosa (*Silent Data Corruption* – SDC), el peor y más peligroso de lo que puede pasar frente a un fallo transitorio. El bit cambiado en el caso del SDC (ítem 7 de la Figura 2) será procesado por la aplicación en ejecución en el sistema

o por el sistema operativo y puede causar consecuencias impredecibles en el comportamiento general del sistema.

En la Figura 3 mostramos seis posibles efectos que en general podemos clasificar un *soft error* en términos de DUE y SDC: la excepción de una instrucción inválida [**17**], un error de paridad en el ciclo de lectura [**17**], una violación en el acceso a la memoria [**17**] y los cambios en una instrucción, en los datos o en una dirección de memoria [**18**] los tres primeros provenientes de DUE y los tres últimos provenientes de SDC.

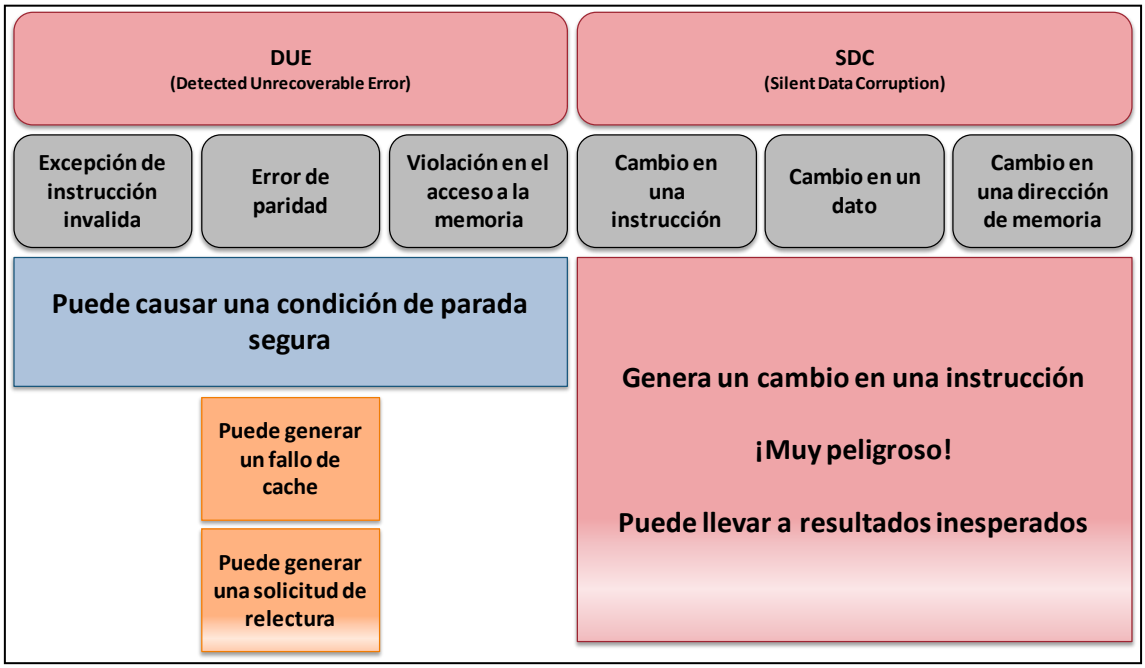

Figura 3 – Posibles efectos de un *soft error*.

2.3.1 Excepción de instrucción inválida/Cambio en una instrucción

Un cambio de un bit en una instrucción de un programa, una vez que haya sido decodificada por el procesador, puede generar una excepción de instrucción inválida si el procesador no es capaz de interpretar la instrucción (DUE).

Pero, si el cambio apenas transforma una instrucción en otra instrucción que sea válida, el *soft error* será propagado y probablemente pasará sin ser notado por el procesador (SDC).

Lo mismo ocurre transformando una instrucción válida en otra instrucción válida por el cambio de un bit, es posible que los parámetros de la instrucción, ubicados en los bytes justo después de la instrucción, no sean adecuados a la ejecución de la instrucción transformada por el cambio, generando una excepción de instrucción invalida (DUE).

2.3.2 Error de paridad durante la lectura

Los errores de paridad durante una operación de lectura pueden pasar principalmente en dispositivos de memoria (la memoria principal del computador, las memorias cache) o en las líneas de comunicación internas.

Es bastante común que las líneas de comunicación internas usen verificación por paridad y que también implementen la posibilidad de retransmisión del dato afectado por el error. El efecto práctico en los computadores de este tipo de situación en un pequeño retraso en la transmisión, pero de forma general el sistema sigue su ejecución sin problemas.

Cuando un *soft error* afecta una posición de memoria, e los componentes afectados poseen verificación por paridad, las consecuencias del error dependerán de donde en la jerarquía de memoria está la porción afectada.

Si el error de paridad ocurre en la memoria principal del computador, es posible que el sistema pueda ser reparado en los casos siguientes:

1. El sistema operativo trabajo con recursos de memoria virtual del procesador y tiene un copia de la porción de la memoria afectada por el fallo guardada en el fichero de paginación. Si la porción afectada de la memoria no ha sido cambiada por cómputo normal antes de la detección del fallo, el sistema operativo puede recuperar del fichero de paginación la porción que contiene la copia de la memoria antes del cambio.
2. El fallo ha afectado la memoria donde estaba el código en binario de la aplicación y hay un fichero con el binario de la aplicación almacenado en otro dispositivo. En este caso, es posible hacer con que el sistema operativo recupere la porción afectada del programa del fichero que contiene el binario original.

Si no hay copias de la porción de memoria afectada que se pueda usar, resta al sistema operativo dos alternativas: hacer una parada segura forzada de la aplicación o informar a la aplicación qu ha pasado un error y dejar que ella intente recuperarse de alguna forma.

Cuando un error de paridad es detectado en una memoria cache, las posibilidades son similares a las que hemos explicado sobre las que afectan la memoria principal: si la porción de la cache afectada no ha sido modificada anteriormente por cómputo normal de la aplicación, el controlador de la memoria puede pedir una nueva copia a la memoria principal (o para el nivel de cache superior) y restaurar el estado anterior de la porción de la cache afectada.

Pero, si la porción de la memoria cache ya estaba cambiada por cómputo normal de la aplicación, el sistema operativo debe o parar la aplicación o le informa sobre el error.

2.3.3 Violación de acceso a la memoria

Un *soft error* que afecte un puntero para la memoria puede hacer con que una aplicación obtenga o almacene datos en regiones de la memoria que no le pertenezcan. Además, la aplicación puede intentar saltar la ejecución para una dirección de memoria no le pertenezca.

Una vez que se identifica que la aplicación está intentando acceder a una región de memoria que o no existe o no le pertenece, el sistema operativo parará la ejecución de la aplicación evitando la propagación del error para otras partes del sistema operativo o para otras aplicaciones.

2.3.4 Cambio de instrucciones, datos y direcciones de memoria

Un fallo transitorio que cambie un bit es capaz de generar un *soft error* que afecte a la operación de un componente cambiando el resultado esperado de este componente para algo que no es esperado.

Este es el caso de los errores que afectan los componentes internos de los procesadores, por ejemplo. Los registros internos, los *pipelines* de ejecución, la unidad de operaciones aritméticas (*Arithmetical Logic Unit* – ALU) y de punto flotante (*Float Point Unit* – FPU) y casi todos los demás componentes de un procesador moderno posee algún tipo de memoria (necesaria para almacenar los resultados intermedios de las operaciones) y algún tipo de línea de transmisión para la comunicación con los otros componentes del procesador. Todas estas memorias auxiliares y estas líneas de comunicación pueden ser afectadas por los fallos transitorios.

Un *soft error* en un componente interno del procesador puede pasar sin ser notado n la operación del sistema si este componente no tiene ningún tipo de protección y si el resultado del cambio inadvertido no genera una violación en el espacio de memoria y también no genera una operación inválida.

La corrupción silenciosa de datos (SDC), el efecto más común de los *soft errors*, no hará con que ninguna aplicación pare su ejecución y solamente tendrá sus efectos notados por los usuarios si el resultado generado por la aplicación al final de su ejecución es significativamente diferente de los resultados normales.

2.4 Robustez y prestaciones

A través de la evaluación del SDC de un sistema o de una aplicación ejecutando en un sistema, podemos evaluar su robustez. Cuanto menor sea la cantidad de SDC que pase un sistema, más robusto esto sistema puede ser considerado.

Entretanto, en [**14**], los autores han evaluado que algunas alternativas de mejorar la robustez de sistemas implicaban en añadir sobrecarga en el tiempo de ejecución para un mismo trabajo.

Entonces, estos autores han propuesto el uso de una nueva métrica, el *Mean Work to Failure* – MWTF. El uso de esta métrica permite la comparación de soluciones de mejoría de robustez basadas en hardware, en software o hibridas.

Ecuación 1 – Fórmula general del Mean Work to Failure – MWTF.

$$MWTF = \frac{\text{cantidad de trabajo completado}}{\text{cantidad de errores encontrados}}$$

El MWTF ha sido elaborado para tener en cuenta el equilibrio entre la reducción en la cantidad de errores encontrados en un sistema que tuvo su robustez mejorada con la reducción en la cantidad de trabajo realizado en un determinado intervalo de tiempo por cuenta de haber añadido algún tipo de sobrecarga para mejorar la robustez.

El la Ecuación 1, donde enseñamos la formula general del MWTF, el concepto de trabajo es abstracto y depende de donde la métrica MWTF es aplicada. El importante en este caso, para comparar diferentes entornos usando el MWTF es usar una medida de trabajo que sea la misma y medida de la misma forma en todos los entornos comparados.

Capítulo 3 Caracterización de la robustez

3.1 Introducción

Para caracterizar la robustez de una aplicación que es ejecutada en una determinada arquitectura se pueden usar dos métodos: el primero basado en la caracterización de la robustez de la arquitectura y después de la aplicación, el segundo basado en una aproximación estadística usando inyección de fallos en la ejecución de la aplicación en la arquitectura.

Hacer la caracterización de la arquitectura es una tarea que implica evaluar la robustez de los componentes usados en la fabricación del procesador y después, calcular la robustez de cada conjunto de componentes responsable para una determinada funcionalidad en el procesador. Así, es posible saber la robustez de todas las operaciones del procesador. Analizando la aplicación a través de las operaciones que va a ejecutar en el procesador hace posible evaluar entonces la robustez de una aplicación en la arquitectura anteriormente caracterizada.

Por tanto, es muy complicado hacer la evaluación de la robustez de una arquitectura sin tener la traza de la misma y sin saber los detalles de su fabricación.

Por otro lado, es posible caracterizar la robustez de una aplicación ejecutando en una arquitectura a través de inyección de fallos.

Esa aproximación en general consiste en, para cada estado de la ejecución de la aplicación en la arquitectura (cada operación que la aplicación solicita al procesador), cambiar un bit de uno de los registros del procesador y después esperar que la aplicación ejecute hasta que termine. Al final de la ejecución, se puede caracterizar el efecto que el fallo inyectado ha causado en la aplicación. Cuando se hayan realizado todas de las probas de todos los bits de todos los registros en todos los estados posibles de la aplicación, es posible evaluar cuantos de los fallos inyectados han afectado la aplicación e inferir, entonces, la robustez.

El problema de la caracterización por inyección de fallos es que la cantidad de ejecuciones de la aplicación necesaria para una evaluación exhaustiva de la robustez es demasiado grande. Por ejemplo, una aplicación de cerca de un segundo puede llegar a tener más de 1 millón de estados y para caracterizarla en una arquitectura con 8 registros de 32 bits cada necesitaría 256 millones de segundos de ejecución (poco menos de tres mil días).

Es posible usar una aproximación estadística, donde se limita la cantidad de estados de la aplicación donde serán inyectados los fallos, seleccionados normalmente de forma aleatoria, con el objetivo de disminuir el tiempo necesario para la caracterización de la robustez de una

aplicación en una arquitectura usando inyección de fallos. En este caso, el bit a ser cambiado y el registro pueden ser elegidos de forma aleatoria, desde que se tenga una distribución significativa que toque todos los registros y bits de forma equivalente a lo largo de las experimentaciones.

Cuantos más experimentos de inyección se realicen para evaluar la robustez de una aplicación ejecutando en una arquitectura, mejor será la precisión de la evaluación.

3.2 Usando inyección de fallos para caracterizar la robustez

Para nuestras caracterizaciones de la robustez de las aplicaciones hemos desarrollado un entorno de inyección de fallos.

El entorno consiste en un conjunto de programas para el sistema operativo Linux. Este conjunto de programas puede ser dividido en dos partes: los que hacen la preparación de una caracterización y los que hacen las ejecuciones con inyección de fallos.

En la Figura 4 enseñamos la fase de preparación de una caracterización. Para esta fase tenemos que informar la aplicación que se quiere caracterizar y la cantidad de fallos que serán inyectados (uno en cada ejecución de la aplicación).

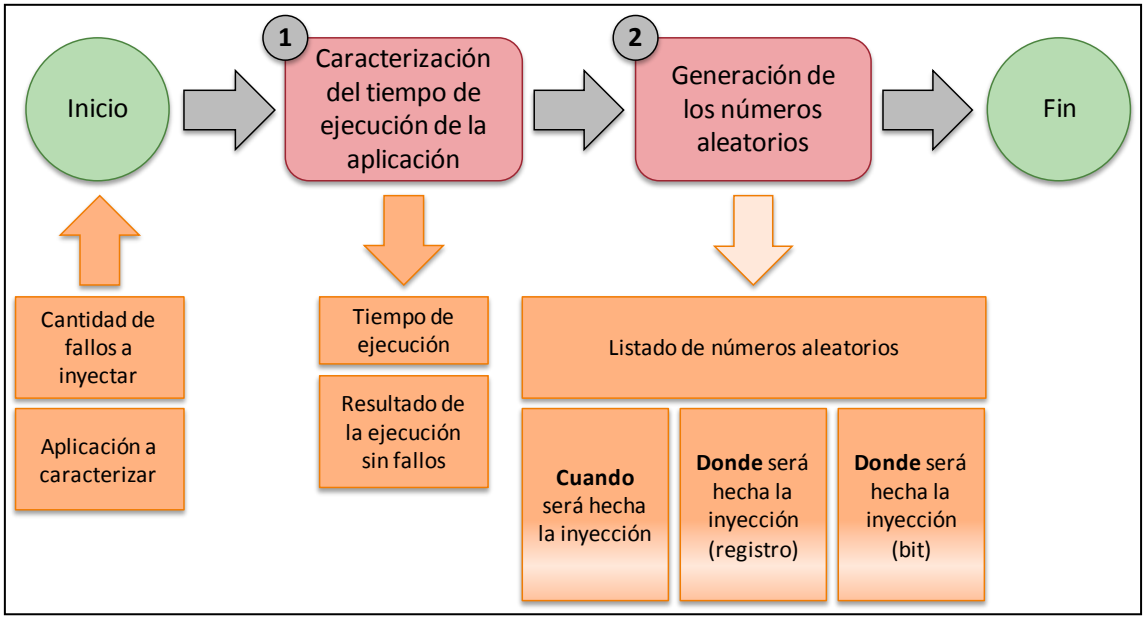

Figura 4 – Fase de preparación de la caracterización de la robustez.

Esta preparación comienza con la caracterización del tiempo de ejecución de la aplicación, ítem 1 de la Figura 4, donde ejecutamos la aplicación 10 veces y obtenemos el menor tiempo de ejecución medido, que se utilizará como el tiempo de ejecución de la aplicación en los cálculos de prestaciones. Además del tiempo de ejecución, también almacenamos el resultado de lo que

es una ejecución correcta de la aplicación (sin presencia de fallos). Este resultado será usado después de cada ejecución con inyección para evaluar si el resultado de la aplicación ha sido afectado o no por el fallo inyectado.

Para terminar la preparación, son generados tres ficheros con números aleatorios suficientes para la cantidad de fallos a inyectar informada. Un generador de números aleatorios es usado para generar tantos números aleatorios (con valores entre cero y uno) cuantos sean la cantidad de inyecciones que se desea hacer en la caracterización. Un fichero contiene los números aleatorios que elegirán el estado de la aplicación donde será hecha la inyección, otro contiene los números aleatorios que elegirán que registro a ser cambiado y el último contiene los números aleatorios que elegirán que bit será cambiado.

En la Figura 5 enseñamos la fase de ejecución con inyección de fallos. Esta fase aprovecha todas las informaciones de entrada y también todas las informaciones generadas por la fase de preparación.

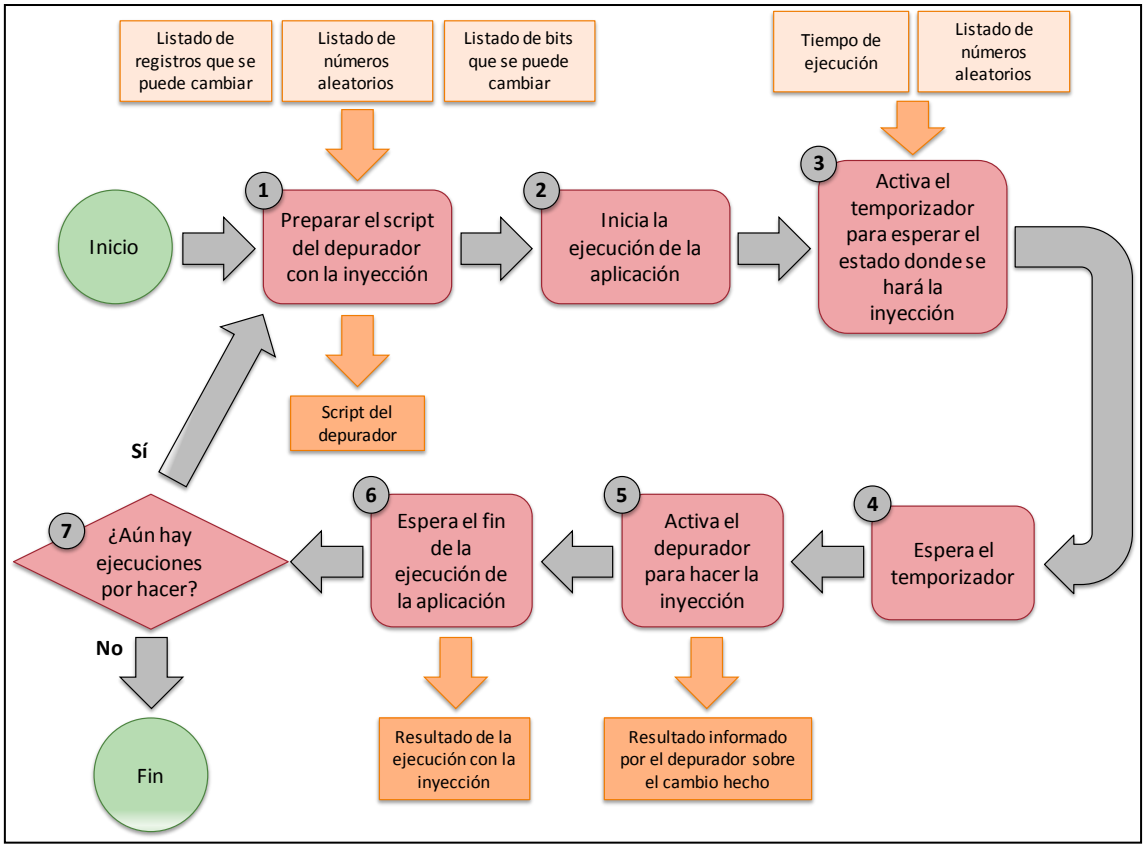

Figura 5 – Ejecuciones con inyección de fallos.

Lo primero que se hace (ítem 1 de la Figura 5) es preparar el script que será usado por el depurador del sistema operativo para la inyección de fallo. Este script necesita la información de que bit será cambiado y en que registro del procesador. Esta información se obtiene cruzando el listado de números aleatorios con los otros dos listados preparados en la fase de preparación de

la caracterización: uno con la lista de registros donde se harán los cambios y otro con la lista de los bits que pueden ser cambiados en los registros.

Después, se inicia la ejecución de la aplicación (ítem 2 de la Figura 5) y se activa un temporizador (ítem 3 de la Figura 5) en paralelo a la ejecución de la aplicación. Este temporizador se basa en el listado de números aleatorios y en el tiempo de ejecución medido en la fase de preparación para determinar cuánto tiempo esperará hasta el momento de la inyección del fallo.

Precisamente cuando terminar la espera del temporizador (ítem 4 de la Figura 5), el depurador del sistema operativo es activado (ítem 5 de la Figura 5) e interrumpe la ejecución de la aplicación. Basándose en el script preparado anteriormente, el depurador hace un cambio de un bit en un registro del procesador y almacena el estado donde se ha hecho el cambio, activando también la continuación de la ejecución de la aplicación.

Hecha la inyección, el entorno espera que finalice la ejecución de la aplicación (ítem 6 de la Figura 5) hasta un límite prefijado en este caso a cinco veces el tiempo de ejecución sin fallos. Si la aplicación no termina antes del límite prefijado (cinco veces el tiempo de ejecución sin fallos), el entorno solicita al sistema operativo que interrumpa y aborte la ejecución de la aplicación.

Los ítems 1 a 7 de la Figura 5 forman un bucle hasta que se hagan tantas inyecciones como se han especificado en la fase de preparación. Como el punto donde se hace la inyección el elegido basado en el tiempo de ejecución, tendremos que garantizar que, mismo que pueda ejecutar más de una tarea simultanea, el experimento deberá ser ejecutado solo en un computador, sin interferencias de otros programas que no pertenezcan al proceso de caracterización.

Un experimento de evaluación de la robustez de una aplicación usando inyección de fallos puede ser clasificado de cuatro formas, según se muestra en la Tabla 1.

Tabla 1 – Clasificación de un experimento de inyección de fallos

Clasificación	Característica
unACE	La aplicación ha ejecutado y terminado. Su resultado es igual al resultado de una ejecución sin presencia de fallos.
pSDC (Detectada por el sistema operativo)	La aplicación ha terminado de forma inesperada.
dSDC (Corrupción de datos)	La aplicación ha ejecutado y terminado, pero su resultado no es igual al resultado de una ejecución sin presencia de fallos.
DUE	El mecanismo de detección de fallos añadido a la aplicación ha detectado el error generado por el fallo inyectado y ha hecho una parada segura.

Sumando los resultados pSDC y dSDC, podemos evaluar la cantidad de veces que la aplicación ha sido afectada por los fallos inyectados sin que se notase. Cuanto menor sea esta suma, mayor será la robustez de la aplicación tras fallos transitorios.

Además de evaluar la robustez de la aplicación en presencia de fallos transitorios, con la inyección de fallos es posible también evaluar los registros que cuando tocados más afectaron la aplicación, o que sitio (función) de la aplicación más ha sufrido con los fallos inyectados.

Para evaluar el MWTF en la caracterización de la robustez, [**14**] propone que, cuando se evalúa una aplicación o un *benchmark*, que sea usado como trabajo en la ecuación del MWTF el inverso del tiempo de ejecución.

Ecuación 2 – Uso del MWTF en la caracterización de la robustez.

$$(1)\quad MWTF = \frac{\text{cantidad de trabajo realizado}}{\text{cantidad de errores encontrados}}$$

$$(2)\quad MWTF = \frac{1}{\text{cantidad de errores encontrados} \times \text{tiempo de ejecución}}$$

$$(3)\quad MWTF = (SDC \times \text{tiempo de ejecución})^{-1}$$

En el ítem 1 de la Ecuación 2 tenemos la forma general del MWTF que hemos presentado en la sesión 2.4 de este documento. Usando la recomendación del los autores en [**14**], en el ítem 2 de la Ecuación 2 presentamos la fórmula general del MWTF que usamos con la metra que hemos usado para el trabajo. Por fin, en el ítem 3 de la Ecuación 2 presentamos la formula que hemos usado en las caracterizaciones para evaluar el MWTF de las aplicaciones que hemos caracterizado con nuestro entorno.

Capítulo 4 Estado del arte

4.1 Introducción

Los fallos transitorios ocurren y una primera forma de tentar minimizar los efectos de los errores generados por estos fallos es a través de la detección de fallos (o la detección de los errores generados por los fallos).

Detectar un fallo implica reconocer un error generado por esto fallo. Una vez reconociendo que un fallo ha pasado, se puede, en un abordaje sencillo, hacer una parada segura del sistema, impidiendo que el error generado por el fallo se propague y afecte el comportamiento del sistema. Esto sería el abordaje más sencillo de gestión de la detección.

En abordajes más elaborados, se puede hacer con que, una vez reconociendo que un fallo ha pasado, se informe a un mecanismo de protección del sistema que ha sido detectado un fallo. Esto mecanismo podrá, entonces, dependiendo de su implementación, volver el sistema hasta un estado anterior al de la detección del fallo y reiniciando su ejecución desde este estado considerado seguro, tolerando el error generado por el fallo. Con eso, no hay la necesidad de hacer una parada segura del sistema y el mismo podrá seguir con su trabajo.

En este estado del arte sobre detección de fallos abordaremos principalmente las iniciativas de detección de fallos con abordaje de parada segura, pero algunos de los trabajos que citaremos también pueden llegar al punto de tolerar los errores provenientes de los fallos transitorios.

4.2 Detección de fallos transitorios

Detección de fallos transitorios, o la detección de los errores generados por fallos transitorios, es un objetivo que en la práctica puede ser alcanzado con el uso de redundancia. Dependiendo de la técnica usada o del nivel donde se ponga la redundancia para detección de errores generados por fallos transitorios, esa redundancia puede ser masiva (incurriendo en sobrecarga elevada o baja eficiencia en el aprovechamiento de las prestaciones del sistema) o moderada [**10**].

Las alternativas de detección de fallos usando redundancia moderada son las que comprometen la eficacia de la detección (no garantizando un 100%) por menor sobrecarga, por mejor eficiencia en las prestaciones o por coste menos elevado de los recursos utilizados.

Las técnicas para detección de errores generados por fallos transitorios pueden ter su mecanismo de detección de fallos basados distintos niveles de un sistema, ilustrados en la Figura 6: en el hardware del sistema, en el sistema operativo, en algún *framework* usado por la

aplicación que se desea dejar más robusta o entonces en la propia aplicación, los tres últimos considerados software.

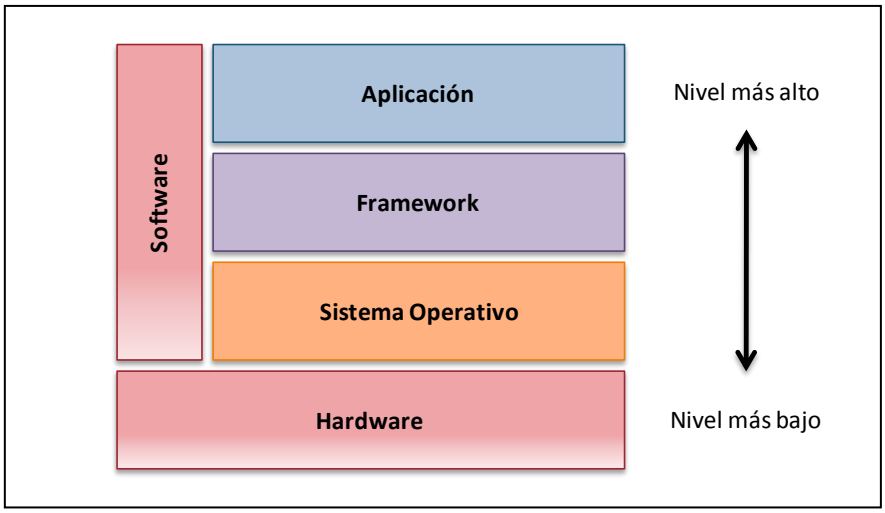

Figura 6 – Niveles donde se puede estar el mecanismo de detección de fallos.

Además, los mecanismos de detección de errores generados por fallos transitorios pueden operar con granularidad baja o alta. La granularidad fina se refiere a la detección de errores rápidamente, justo después de que el fallo tenga generado el error. La granularidad alta en la detección de errores generados por fallos transitorios se basa en algunos procedimientos de control, sea sobre la ejecución del programa (por ejemplo, comparación de los resultados finales obtenidos en dos versiones distintas de un mismo programa) o su parte significativa (por ejemplo, verificando algunos resultados parciales de un determinado segmento de código) [**17**].

De modo general, cuanto más bajo el nivel donde se pone el mecanismo de detección de fallos, más transparente esto mecanismo es para la aplicación que ejecuta en el sistema y más fina es su granularidad.

4.2.1 Detección basada en hardware

Que un hardware sea más robusto contra fallos transitorios es lo que los administradores de sistemas y usuarios de las aplicaciones más desean. Entretanto, este hardware más robusto normalmente es más caro y no logra las prestaciones de los hardwares menos robustos.

Las primeras formas de dejar un hardware más robusto contra fallos transitorios han trabajado en los materiales usados en la fabricación del hardware.

Hoy, es cierto que las velocidades de los procesadores pueden ser bastante altas, llegando a miles de megahercios, pero cuanto más se presiona la subida de la velocidad de los procesadores, menos robustos ellos quedan frente a fallos transitorios. Poe ejemplo, una forma

de tener un hardware más robusto es limitar la velocidad de los procesadores en niveles considerados más seguros, comprometiendo prestaciones por robustez.

Los fallos transitorios que afectan las memorias de los computadores se tornaran tan frecuentes a medida que su densidad y velocidad fueran aumentando que han estimulado el uso de mecanismos de redundancia en hardware para detección (CRC) y después para detección y corrección (ECC) de errores en las memorias de forma transparente al resto del sistema.

En nuestro trabajo no trabajaremos con fallos en las memorias y no analizaremos los mecanismos de detección de errores causados por fallos transitorios en memorias. Nuestro trabajo está enfocado en los fallos transitorios que pueden pasar dentro del procesador, en su memoria interna (los registros) o en sus circuitos de cómputo.

4.2.1.1 Redundancia modular

Una de las formas detectar los errores generados por fallos transitorios usando el hardware es a través de redundancia modular.

Con la redundancia modular dupla (DMR – *Dual Modular Redundancy*), dos procesadores trabajan de forma separada haciendo el mismo cómputo y el resultado del cómputo de los dos es comparado por un mecanismo de verificación. Es coste de construcción de un mecanismo que use DMR es bastante elevado. Además de garantizar que los dos procesadores hacen lo mismo exactamente al mismo tiempo (*lockstep*), se usan dos procesadores para generar un resultado valido de ejecución (un 50% de eficiencia si entendemos que los dos procesadores podrían estar repartiendo trabajo para mejorar las prestaciones del sistema) [**19**].

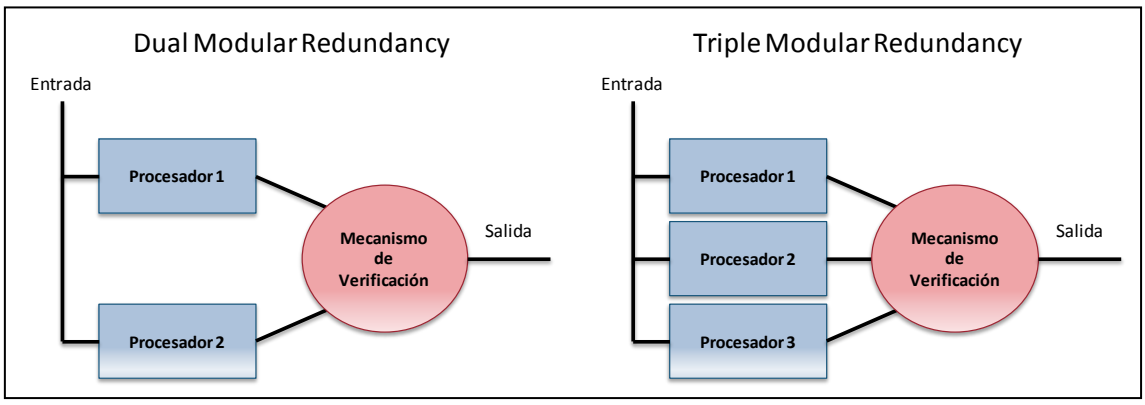

Figura 7 – Redundancia modular con dos y tres procesadores.

Si un sistema con redundancia modular usa tres procesadores (TMR – Triple Modular Redundancy), como mostrado juntamente con la DMR en la Figura 7, el mecanismo de detección trabajará haciendo una votación comparando los tres resultados generados. Con esto, no solamente el sistema logra detectar errores pero también puede saber cuál de los procesadores ha fallado y así, seguir la ejecución del sistema con los otros resultados que ya se

sabe ser buenos. Usando tres procesadores, la eficiencia del sistema cae para cerca de 33,3%, aún menor que la del DMR [**19**].

Actualmente, tanto HP cuanto IBM son fabricantes que tienen líneas de computadores que trabajan con redundancia modular disponibles al mercado. Sus productos fueran desarrollados para el seguimiento de computación de misión crítica donde la fiabilidad es más importante que las prestaciones.

En la redundancia modular, los dos procesadores trabajan ejecutando el mismo programa de forma completamente independiente, reciben los mismos datos de entrada de la memoria y sus salidas, después de verificadas, son almacenadas en la memoria como si solamente hubiera un procesador.

4.2.1.2 Redundancia usando varias tareas de un mismo procesador

La evolución de los procesadores en búsqueda de mejores prestaciones ha provocado el aparecimiento de un recurso llamado *Simultaneous Multithreading* (SMT) que permite que un procesador superescalar comparta sus recursos internos, que muchas veces quedaban ociosos, con dos o más tareas de ejecución independientes.

Usando dos tareas de un mismo procesador para ejecutar un mismo programa, se buscaba tener una solución menos costosa que la redundancia modular.

En el trabajo presentado por [**20**] se usó un simulador evaluar las prestaciones de una solución basada en la técnica de SMT donde han cambiado el proyecto de un procesador usado como base para que este pasase a trabajar con dos tareas ejecutando de forma redundante distintas copias de un mismo programa, técnica que llamó *Simultaneous and Redundantly Threaded* (SRT).

De forma distinta de la redundancia modular, el uso de tareas redundantes en un mismo procesador no trabaja con *lockstep*, mejorando las prestaciones de un conjunto de aplicaciones (Spec95) en 16% en promedio segundo presentado en [**20**]. Todavía, no hemos visto procesadores comercialmente disponibles con el recurso de SMT.

4.2.1.3 Redundancia usando ejecuciones en procesadores distintos

Aún trabajando con cambios en el procesador, [**21**] presentó un trabajo donde proponía una técnica llamada *Redundant Multithreading* (RMT), parecida con la SRT, pero que no usaba tareas de un procesador superescalar.

Con el RMT, [**21**] propuso que un procesador podría trabajar con dos tareas de forma redundante para detección de errores generados por fallos transitorios mismo sin tener dos tareas diseñadas solamente para eso. Pero la degradación de las prestaciones que logro con el

RMT quedó en un 32% en promedio, resultado peor que el trabajo de [**20**] con el SMT que lograba 21% de degradación en relación a la ejecución sin detección de errores.

Intentando mejorar sus resultados, [**21**] entonces presentó una nueva propuesta, aplicando el mecanismo RMT en un entorno con dos procesadores. La técnica llamada *Chip-Level Redundant Threading* (CRT) mejora las prestaciones en comparación a la ejecución con *lockstep* en un 13% en promedio, obteniendo sus mejores resultados en experimentos con cargas de trabajo con varias tareas simultáneas.

Todavía, mismo logrando mejoras en comparación a los mecanismos de redundancia modular, las técnicas RMT y CRT necesitan de cambios en la arquitectura de los procesadores e igual que la técnica SMT, un han sido implementadas comercialmente.

4.2.1.4 Redundancia usando ejecuciones núcleos distintos

Siguiendo la línea de proponer cambios en arquitecturas de procesadores para hacer la detección de errores generados por fallos transitorios, un trabajo de [**22**] presento una propuesta de una solución basada en el DMR pero usando núcleos distintos de un mismo procesador para ejecutar las copias redundantes del programa de forma configurable. Se puede usar el entorno propuesto con o sin redundancia, de forma que la sobrecarga de la redundancia solamente afecte a lo que los usuarios crean que sea importante.

En este mismo trabajo, [**22**] evidencia la dificultad en implementar estos cambios de arquitectura propuestos en sistemas reales. Además evidencia que cuando usado con redundancia, el entorno propuesto añade una sobrecarga en la ejecución que puede llegar hasta 100%, pero que seguramente garantizaría 100% de detección de los errores provocados por los fallos transitorios.

4.2.2 Detección basada en software

En comparación a las alternativas de detección de errores generados por fallos transitorios basadas en hardware, las alternativas basadas en software tienes la ventaja de poder ser usadas en los entornos reales con procesadores actuales, sin necesidad de cambios en la arquitectura de los procesadores.

Pero si las soluciones de detección basadas en redundancia en el hardware tienen una tendencia a detectar todos los errores generados por fallos transitorios, las soluciones basadas en software prácticamente no lo pueden hacer lo mismo.

La inserción de la redundancia para detección de los errores generados por los fallos transitorios en la capa de software puede estar en el sistema operativo, en un *framework* intermedio o en la propia aplicación.

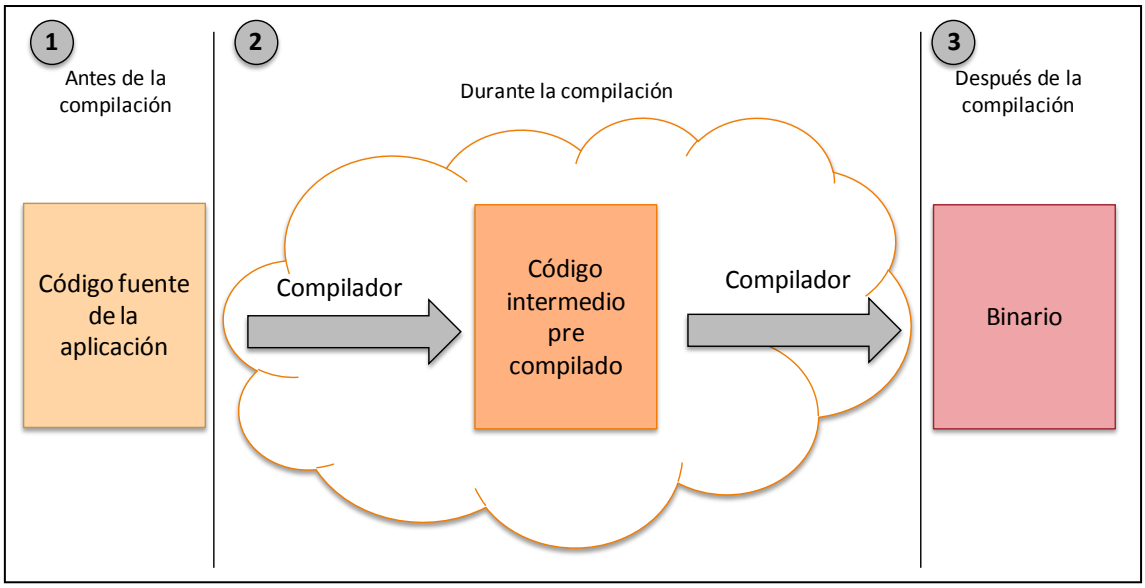

Figura 8 – Categorización de donde y cuando si añade la redundancia en las aplicaciones.

Presentaremos algunas de las soluciones propuestas que añaden la redundancia en la aplicación, que podemos categorizar en tres formas básicas, presentadas en la Figura 8:

1. Añadir la redundancia antes de la compilación de la aplicación, analizando y cambiando el código fuente de la misma;
2. Añadir la redundancia durante la compilación de la aplicación, analizando el código intermedio generado por el compilador y cambiando él antes de la generación del binario;
3. Añadir la redundancia después de la compilación de la aplicación, analizando y cambiando el binario de la misma.

4.2.2.1 Añadiendo la redundancia antes la compilación

Trabajar con el código fuente de la aplicación que se pretende tornar más robusta frente a los efectos de los fallos transitorios tiene como principal ventaja que, cambiando el código fuente, se puede tornar una aplicación más robusta independiente de la arquitectura donde ella va ejecutar.

En [**13**], los autores describen una técnica de software que permite la detección de los errores de generados por fallos transitorios. El mecanismo de detección se basa en un conjunto de reglas que permiten la transformación del código fuente de la aplicación en otro que genera otro programa con las mismas funcionalidades que el anterior, pero que también es capaz de identificar posibles cambios de bit en áreas de la memoria y en los registros internos del procesador.

Las reglas, propuestas, de forma general, objetivan añadir redundancia a los cómputos de dos datos usados en la aplicación, además de verificar el flujo de la aplicación (si no ha pasado un

salto para una región de forma inadvertida) y también añadir testes redundantes en las estructuras condicionales.

En el ejemplo presentado en el trabajo publicado, los autores han logrado una mejoría de 76,1% en la robustez de la aplicación con una sobrecarga en torno dos 144%, el que implicaría en un MWTF de 1,72 veces mejor que la aplicación sin detección de fallos.

4.2.2.2 Añadiendo la redundancia durante la compilación

Como los errores generados por los fallos transitorios en un procesador terminan por cambiar de forma inadvertida el contenido de un registro del procesador o el resultado de un cómputo interno del procesador que será almacenado posteriormente en un registro, las técnicas de detección de errores generados por fallos transitorios en general añaden la redundancia en las operaciones a nivel de los registros de los procesadores.

El trabajo presentado por [**10**] introduce la idea de detección de errores por duplicación de instrucciones (*Error Detection by Duplicated Instructions* – EDDI), una aportación que propone la duplicación de las instrucciones del programa pero usando registros diferentes de un procesador superescalar, conforme ilustrado en la Tabla 2, durante la compilación del programa.

Tabla 2 – Ejemplo de redundancia con EDDI.

Código fuente original	Código fuente duplicado

ADD R3, R1, R2; R3 ← R1 + R2	ADD R3, R1, R2; R3 ← R1 + R2
	ADD R23, R21, R22; R23 ← R21 + R22
	BNE R3, R23, ErrorHandler

Por tratarse de un trabajo basado en procesadores superescalares, los autores acreditaban que, mismo con toda la duplicación realizada para añadir la redundancia, la sobrecarga sería inferior a 100% por la posibilidad de tener paralelismo a nivel de instrucciones dentro del procesador. En verdad, usando un procesador con posibilidad de procesar hasta cuatro instrucciones a la vez, usando un conjunto de ocho aplicaciones distintas, el EDDI ha logrado una sobrecarga de 61,5% en promedio, llegando a un mínimo de 13,3% en su mejor caso y a un máximo de 105,9% en su peor caso.

La mejoría en la robustez del conjunto de experimentos presentados por [**10**] obtuvo un 43,9% en promedio, llegando a 65,4% en su mejor caso. Extrapolando el MWTF de los experimentos presentados, dado que no ha sido calculado por los autores del trabajo original, creemos la propuesta EDDI lograría 1,22 en promedio, con 0,74 en su peor caso y 1,63 en su mejor caso, normalizado en relación a los resultados obtenidos con la aplicación sin redundancia. Los resultados mínimos, medios y máximos del EDDI están graficado en la Figura 9.

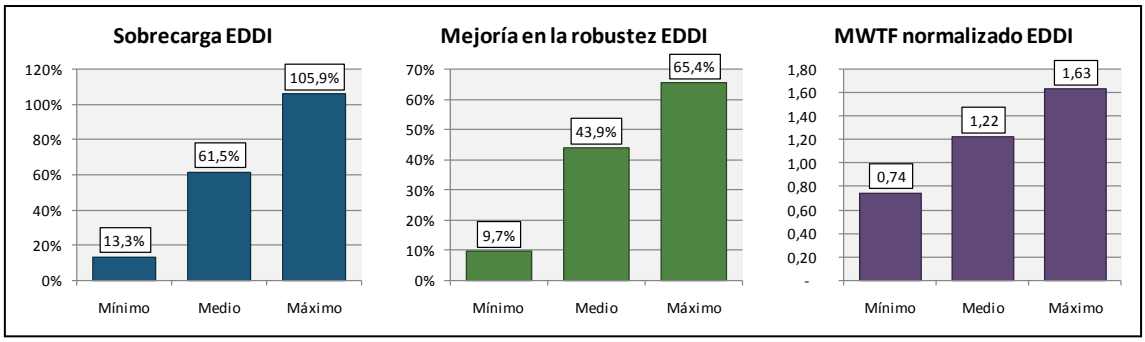

Figura 9 – Sobrecarga, mejoría en la robustez y MWTF obtenido con el EDDI.

El mejor MWTF que el EDDI ha obtenido no ha sido con su mejor robustez y si con su menor sobrecarga. La mejor robustez ha logrado un MWTF normalizado de 1,52.

Basándose en el EDDI, los autores de [**11**] han propuesto SWIFT, que también hace las transformaciones para añadir la redundancia durante la compilación de un programa. Diferentemente de la propuesta EDDI, que trabajaba con el procesador R10000 de la MIPS, los de la propuesta SWIFT han trabajado con el procesador Itanium 2 de Intel (arquitectura IA64). La principal diferencia de los dos trabajos es que el SWIFT ha propuesto cambios en el EDDI para mejorar las prestaciones disminuyendo la sobrecarga añadida por la redundancia de cómputo. Estos cambios intentaban eliminar las penalizaciones en el acceso a la memoria por las instrucciones redundantes.

La propuesta SWIFT añade la redundancia en el código intermedio usado por el compilador, justo antes de la fase de asignación de registros. Además, algunas optimizaciones del compilador que podrían afectar los mecanismos de detección añadidos fueran suprimidas del compilador.

En los resultados presentados por [**11**], la sobrecarga de la redundancia añadida obtuvo un valor de 41% en promedio, usando 30 aplicaciones distintas de *benchmarks* como SPEC95, SPEC2000 y MediaBench.

Además, la propuesta SWIFT ha logrado una mejoría en la robustez de las aplicaciones que ha probado en un 68,4%.

Los autores de [**11**], además de la propuesta SWIFT, han propuesto dos adiciones a la propuesta EDDI y han comparado sus resultados con la propuesta SWIFT. Los resultados de esta comparación están graficados en la Figura 10.

Mismo disminuyendo la sobrecarga en relación a la propuesta EDDI mejorada, la propuesta SWIFT ha obtenido una mejoría de la robustez de las aplicaciones probadas menor que la EDDI

mejorada, fato que ha dejado su MWTF en normalizad en 2,25, valor menor que el 2,47 obtenido por la propuesta EDDI mejorada en el mismo conjunto de experimentos.

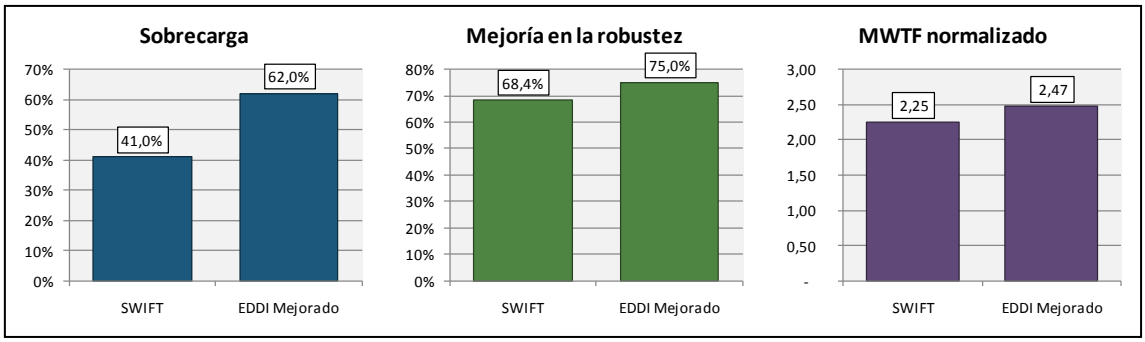

Figura 10 – Sobrecarga, mejoría en la robustez y MWTF de SWIFT y EDDI mejorado.

En ESoftCheck [**12**], los autores se basaran en el trabajo de [**11**] y proponen reglas de optimización para la fase de compilación de una aplicación, donde entre otras cosas, se intenta minimizar la cantidad de testes realizados en la comparación de la ejecución original con su redundancia. Además, en el trabajo propuesto por [**12**], las optimizaciones poseen ayustes, donde si puede comprometer prestaciones por mejorar la robustez o, al revés, se puede comprometer robustez para mejorar las prestaciones.

Este equilibrio se ayusta por dos parámetros: el grado de fiabilidad que se cree que una operación tiene y por la distancia entre las comparaciones de la ejecución original con la redundante.

Los autores han probado su trabajo en un entorno con un procesador Pentium 4 de Intel usando aplicaciones del *benchmark* SPEC2000 y han comparado su propuesta (ESoftCheck) con una basada en el trabajo de [**11**] que llamaremos FullRep (SWIFT).

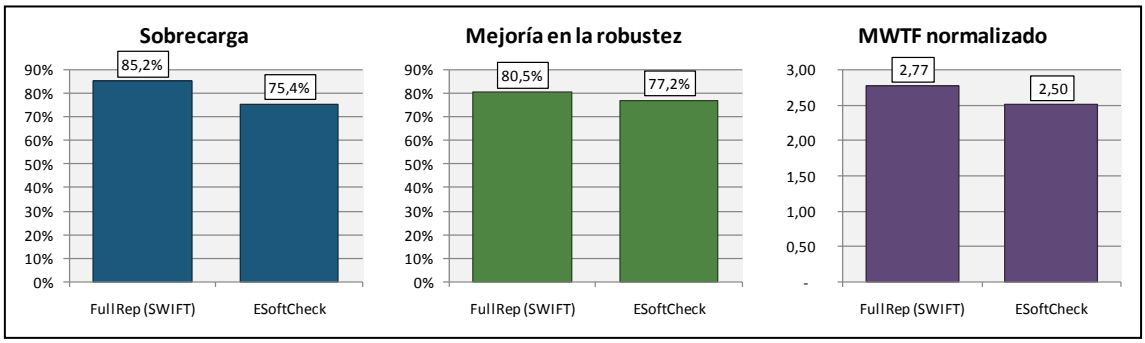

Figura 11 – Sobrecarga, mejoría en la robustez y MWTF de ESoftCheck y FullRep (SWIFT)

En la Figura 11, donde hemos graficado los resultados en promedio presentados por [**12**], notamos que el ESoftCheck si logra disminuir la sobrecarga en relación a la propuesta con la cual se comparaban, pero ha empeorado un poco en la robustez. La perdida en la robustez del ESoftCheck ha sido suficiente para que, cuando calculamos el MWTF normalizado en relación

a una ejecución sin redundancia, lograse un resultado bueno pero menor que el FullRep (SWIFT).

4.2.2.3 Añadiendo la redundancia después de la compilación

Las propuestas de añadir la redundancia después de la compilación del programa, basándose solamente en su binario, tienen la ventaja en relación a las otras dos alternativas cuando no si tiene disponible el código fuente de la aplicación.

En [**9**], los mismos autores de la propuesta SWIFT [**11**] presentan Spot, una técnica para añadir la redundancia propuesta anteriormente en SWIFT pero de esta vez, a través de instrumentación en dinámico del binario de la aplicación.

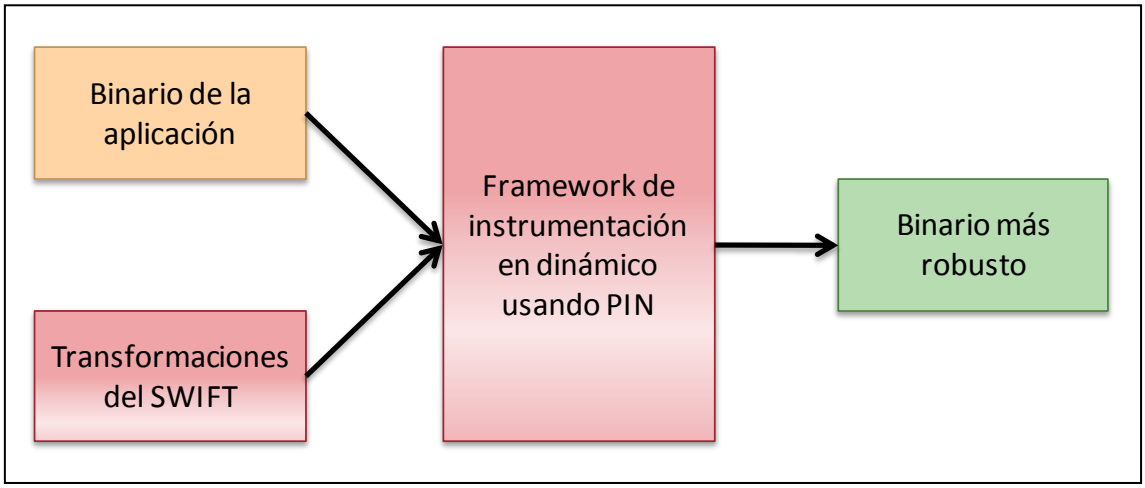

Figura 12 – Diagrama del *framework* Spot.

En la Figura 12, adaptada de [**9**], enseñamos de forma general como son las interacciones entre el binario original de la aplicación, el SWIFT y el Spot.

Spot ha sido diseñada para trabajar con procesadores de la arquitectura x86 y ha sido probada con algunas de las aplicaciones del *benchmark* SPEC2000 compiladas para ejecución en un entorno de 32 bits con procesador Pentium D de Intel. Los resultados obtenidos por Spot muestran que la propuesta ha sido capaz de mejorar la robustez de las aplicaciones en un 68,1% en promedio, pero con una sobrecarga elevadísima de 319% en promedio, lo que hice con que su MWTF normalizado quedase en 0,75 (por debajo de 1), o sea, peor que la ejecución de las aplicaciones sin redundancia.

En un segundo momento, los autores de [**9**] proponen una análisis más profunda sobre en qué partes de las aplicaciones poner la redundancia y que registros del procesador proteger, tarea hecha do forma no automatizada. Para elegir qué partes y que registros, los autores han caracterizado cada una de las aplicaciones frente a fallos transitorios. Con base en esta caracterización, se ayustaba el Spot para no instrumentar todas las aplicaciones pero solamente

las partes que más provocaban corrupción de datos. Con esta evaluación manual caso a caso, el Spot ha logrado resultados con mejoría en la robustez casi igual que antes, pero con sobrecarga significativamente inferior, obteniendo un MWTF normalizado que variaba de 1,9 hasta 17,79.

4.3 Donde y como hemos elegido añadir la redundancia

Las propuestas de añadir redundancia verificando el uso de registros se muestran potencialmente efectivas para procesadores con grande cantidad de registros (Power, SPARC y IA64 por ejemplo, con 32, 32 y 128 registros de uso general respectivamente).

Si trasladamos esta perspectiva hacia entornos minimalistas, con procesadores extremamente especializados, que fue el caso presentado en [**13**], duplicar las operaciones del programa basadas en los registros se muestra un problema, por no haber tantos registros de uso general disponibles para añadir el cómputo redundante.

El problema es que, de los procesadores usados en los grandes centros de supercomputación, destacamos en la lista de los Top 500 [**15**] (de enero de 2010) que apenas 50 supercomputadores no eran basados en procesadores Intel o AMD con 16 registros de uso general, conforme podemos observar en los datos de la Tabla 3.

Vale resaltar que los procesadores de las arquitecturas Intel EM64T y AMD x86_64, cuando usados con aplicaciones compiladas con compatibilidad para ejecución en entornos de 32 bits poseen solamente 8 registros de uso general, limitando aún más la posibilidad de trabajar con duplicación a nivel de los registros del procesador sin afectar las prestaciones.

Tabla 3 – Distribución de las arquitecturas de procesadores de la lista Top 500.

Arquitectura del procesador	Cantidad de supercomputadores en la lista Top 500
Intel EM64T	401 (80,2%)
AMD x86_64	49 (9,8%)
Power	42 (8,4%)
Intel IA-64	5 (1,0%)
Sparc	2 (0,4%)
NEC	1 (0,2%)

En la Figura 13, graficada con base en los datos de la Tabla 3 queda aún más evidente el dominio de estas arquitecturas (de Intel y AMD) que no tienen tantos registros disponibles a los usuarios para añadir redundancia de cómputo para detección de errores generados por fallos transitorios, con 90% de representatividad.

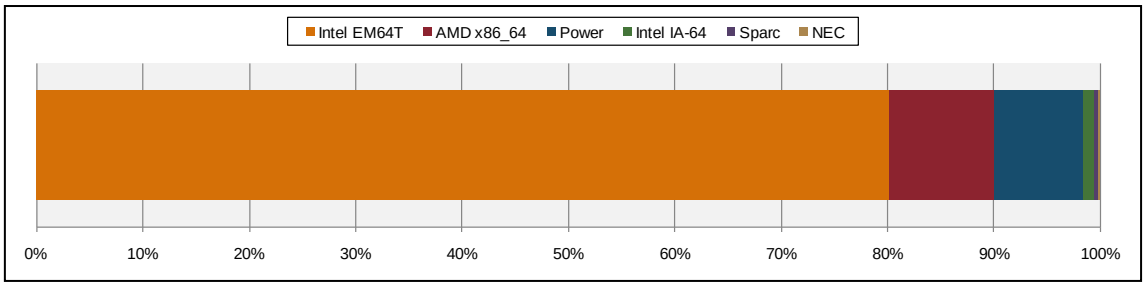

Figura 13 – Arquitectura de los procesadores de los supercomputadores de la lista Top 500.

Buscando una alternativa a la propuesta presentada por [**13**], nuestro objetivo en este trabajo es proponer una forma de cambiar el código fuente de las aplicaciones, mejorando la robustez de las mismas, pero sin añadir sobrecarga que comprometa el equilibrio entre la robustez y las prestaciones, de forma que el MWTF normalizado obtenido sea equivalente o mejor que las propuestas anteriormente presentadas.

Capítulo 5 Mejorando la robustez

5.1 Introducción

Como se expuso anteriormente en este documento, el objetivo de este trabajo es mejorar la robustez de las aplicaciones frente a fallos transitorios realizando transformaciones en el código fuente de la aplicación.

Consideramos que usando la duplicación en el código fuente, aprovecharemos el mejor el uso de registros de un procesador en la ejecución de una aplicación, que serán alocados a los cómputos por el compilador.

También, como las duplicaciones a nivel del binario de la aplicación basándose en el uso de los registros del procesador generan cantidad significativa de DUE falsos [**12**], consideramos que usando la duplicación en el código fuente evitaremos los falsos positivos de detección de errores generados por fallos que no afectarían el resultado final del computo.

Para ayudar en el entendimiento de cómo las reglas propuestas cambian el código fuente de una aplicación, hemos elegido una función que representa un código de una multiplicación de matrices como caso base. El código de la función original sin cambios está en la Figura 14.

```
void mult()
{
    int i, j, k;
    double S;
    for ( i = 0 ; i < matrixSize ; i ++ )
        for ( j = 0 ; j < matrixSize ; j ++ )
        {
            S = 0;
            for ( k = 0 ; k < matrixSize ; k ++ )
            {
                S += A (i, k) * B (k, j);
            }
            C (i, j)= S;
        }
}
```

Figura 14 – Código fuente ejemplo sin detección de fallos.

La función contiene 4 variables locales (3 de tipo punto fijo y una de tipo punto flotante) y tres bucles.

5.2 El caso base: la regla DIC

El acrónimo DIC proviene de *Duplicate Intermediate Computation* (duplicación del cómputo intermedio).

Este caso base hemos extrapolado basándonos en los trabajos presentados en el Capítulo 4. Este abordaje contiene 5 reglas básicas que, cuanto aplicadas a un código fuente, hace con que la especificación del programa contenga redundancia en todas las variables locales de la parte del

programa que se está añadiendo la redundancia y que todos los cómputos que usan estas variables serán duplicados y sus resultados serán verificados comparándolos con los resultados obtenidos con las variables originales del programa.

Para transformar un código fuente usando esta regla hay que seguir los cinco pasos que siguen:

1. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
2. Para cada variable local declarada, crear una copia;
3. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;
4. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
5. Para cada uso de una variable local previamente declarada, añadir antes del uso una verificación si las dos copias de la variable son iguales.

La Figura 15 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla DIC.

```
static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                if (j != j_duplicate) FaultDetected ();
                if (k != k_duplicate) FaultDetected ();
                if (i != i_duplicate) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            if (S != S_duplicate) FaultDetected ();
            if (j != j_duplicate) FaultDetected ();
            if (i != i_duplicate) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 15 – Código fuente ejemplo con la regla DIC.

Aplicando las reglas de transformación propuestas en DIC, hemos observado que, dependiendo de la optimización configurada en el compilador, la eficiencia del mecanismo de detección de errores añadido puede variar.

5.3 Nuevas reglas de transformación propuestas

Intentando mejorar la eficiencia de la detección de errores generados por fallos, hemos propuesto básicamente dos alternativas de reglas de transformación de código: una basada en una característica del lenguaje C (sesión 5.3.1) y otra donde proponemos la creación de un acumulador de errores (sesiones 5.3.2, 5.3.3, 5.3.4 y 5.3.5).

Por cuenta de los efectos que las distintas estrategias de optimización que hacen el compilador dependiendo de lo que pide el usuario cuando compila un programa, en las distintas estrategias usando acumuladores de errores hay variaciones sencillas en como si hace la verificación de las copias del cómputo.

5.3.1 Regla V

Investigando sobre las optimizaciones de los compiladores, encontramos que en el lenguaje de programación C hay una palabra clave (*volatile*) que, cuando usada en la declaración de una variable, enseña al compilador que tenga más cuidado en la optimización relativa a esta variable.

El acrónimo V de esta regla proviene de *Volatile* (volátil).

Para transformar un código fuente usando la regla de transformación V hay que seguir los cinco pasos que siguen:

1. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
2. Para cada variable local declarada, crear una copia y colocar el calificador *volatile* en las dos declaraciones (la original y la copia);
3. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;
4. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
5. Para cada uso de una variable local previamente declarada, añadir antes del uso una verificación si las dos copias de la variable son iguales. En caso negativo, llama la función *FaultDetected*.

La Figura 16 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla V.

```
static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    volatile int i, j, k;
    volatile int i_duplicate, j_duplicate, k_duplicate;
    volatile double S;
    volatile double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                if (j != j_duplicate) FaultDetected ();
                if (k != k_duplicate) FaultDetected ();
                if (i != i_duplicate) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            if (S != S_duplicate) FaultDetected ();
            if (j != j_duplicate) FaultDetected ();
            if (i != i_duplicate) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 16 – Código fuente ejemplo con la regla DICV.

5.3.2 Reglas EAD/EDI

Con el objetivo de reducir la cantidad de testes hechos antes de cada sentencia de código fuente que usaba variables locales, hemos diseñado un acumulador de errores.

Los acrónimos EAD/EDI de estas reglas provienen de *Error Accumulator Double* y *Error Accumulator Integer* (acumulador de errores de tipo punto flotante/acumulador de errores de tipo punto fijo).

El acumulador de errores es una variable global de la aplicación, usada para acumular la diferencia entre una variable y su copia.

En este primer diseño, teníamos o un acumulador de errores de tipo punto flotante (*double*) o un acumulador de errores de tipo punto fijo (*int*).

Para transformar un código fuente usando las reglas EAD y EAI hay que seguir los siete pasos que siguen:

1. Declarar una variable global de tipo *double* (EAD) o *int* (EAI) en la aplicación;
2. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
3. Para cada variable local declarada, crear una copia;
4. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;

5. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
6. Para cada uso de una variable local previamente declarada, añadir antes del uso una declaración que añade al acumulador de errores la diferencia entre las dos copias de la variable usada;
7. Insertar una verificación en el acumulador de errores después de las acumulaciones añadidas por el ítem 6 y antes del uso de las variables.

La Figura 17 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla EAD y la Figura 18 presenta el código fuente transformado con la regla EAI.

```
static double dicea_ERROR = 0.0f;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((double) j - (double) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) k - (double) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) i - (double) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            dicea_ERROR = dicea_ERROR + ((double) S - (double) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) j - (double) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) i - (double) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 17 - Código fuente ejemplo con la regla EAD.

```
static int dicea_ERROR = 0;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((int) j - (int) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) k - (int) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) i - (int) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            dicea_ERROR = dicea_ERROR + ((int) S - (int) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) j - (int) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) i - (int) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 18 – Código fuente ejemplo con la regla EAI.

5.3.3 Reglas IDD/IDI

Aún con la propuesta del acumulador de errores, hemos intentado mejorar la efectividad del mecanismo de detección de fallos cuando usando la optimización máxima del compilador.

Para esto, hemos transformado la declaración de acumular la diferencia de una variable y su copia por dos sentencias: una incrementando el acumulador de errores con el contenido de la variable original y otra decrementando el acumulador de errores con el contenido de la copia de la variable.

Los acrónimos IDD/IDI de estas reglas provienen de *Increase and Decrease Double* y *Increase and Decrease Integer* (incrementando y decrementando con tipo punto flotante/incrementando y decrementando con tipo punto fijo).

Para transformar un código fuente usando las reglas IDD e IDI hay que seguir los siete pasos que siguen:

1. Declarar una variable global de tipo *double* (IDD) o *int* (IDI) en la aplicación;
2. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
3. Para cada variable local declarada, crear una copia;
4. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;
5. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
6. Para cada uso de una variable local previamente declarada, añadir antes del uso dos declaraciones:
    a. Una que incrementa el acumulador de errores con el contenido de la variable original;
    b. Otra que decrementa el acumulador de errores con el contenido de la copia de la variable usada;
7. Insertar una verificación en el acumulador de errores después de las acumulaciones añadidas por el ítem 6 y antes del uso de las variables.

La Figura 19 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla IDD y la Figura 20 presenta el código fuente transformado con la regla IDI.

```
static double dicea_ERROR = 0.0f;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((double) j);
                dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) k);
                dicea_ERROR = dicea_ERROR - ((double) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) i);
                dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            dicea_ERROR = dicea_ERROR + ((double) S);
            dicea_ERROR = dicea_ERROR - ((double) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) j);
            dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) i);
            dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 19 – Código fuente ejemplo con la regla IDD.

```
static int dicea_ERROR = 0;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((int) j);
                dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) k);
                dicea_ERROR = dicea_ERROR - ((int) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) i);
                dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
            }
            dicea_ERROR = dicea_ERROR + ((int) S);
            dicea_ERROR = dicea_ERROR - ((int) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) j);
            dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) i);
            dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
            C (i, j) = S;
        }
}
```

Figura 20 – Código fuente ejemplo con la regla IDI.

5.3.4 Reglas TOD/TOI

Siguiendo con la propuesta del acumulador de errores, hemos intentado reducir la cantidad de testes hechos en el acumulador de errores para mejorar las prestaciones de la aplicación con el mecanismo de detección de fallos.

Para esto, reducido la cantidad de testes en el acumulador de errores para una por cada seguimiento de código.

Los acrónimos TOD/TOI de estas reglas provienen de *Testing Once Double* y *Testing Once Integer* (testando solamente una vez con tipo punto flotante/testando solamente una vez con tipo punto fijo).

Para transformar un código fuente usando las reglas TOD y TOI hay que seguir los pasos que siguen:

1. Declarar una variable global de tipo *double* (TOD) o *int* (TOI) en la aplicación;
2. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
3. Para cada variable local declarada, crear una copia;
4. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;
5. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
6. Para cada uso de una variable local previamente declarada que aún no tenga sido testeada en el seguimiento de código actual, añadir antes del uso dos declaraciones:
    a. Una que incrementa el acumulador de errores con el contenido de la variable original;
    b. Otra que decrementa el acumulador de errores con el contenido de la copia de la variable usada;
7. Al final de un seguimiento código, si han tocado el acumulador de errores con el ítem 6 en el seguimiento actual:
    a. Insertar nuevamente el incremento y el decremento de todas las variables que tengan sido testeadas en el seguimiento de código actual;
    b. Insertar una verificación en el acumulador de errores.

La Figura 21 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla TOD y la Figura 22 presenta el código fuente transformado con la regla TOI.

```
static double dicea_ERROR = 0.0f;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((double) j);
                dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) k);
                dicea_ERROR = dicea_ERROR - ((double) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) i);
                dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) j);
                dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) k);
                dicea_ERROR = dicea_ERROR - ((double) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((double) i);
                dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
            }
            dicea_ERROR = dicea_ERROR + ((double) S);
            dicea_ERROR = dicea_ERROR - ((double) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) j);
            dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) i);
            dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
            C (i, j) = S;
            dicea_ERROR = dicea_ERROR + ((double) S);
            dicea_ERROR = dicea_ERROR - ((double) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) j);
            dicea_ERROR = dicea_ERROR - ((double) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((double) i);
            dicea_ERROR = dicea_ERROR - ((double) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
        }
}
```

Figura 21 – Código fuente ejemplo con la regla TOD.

```
static int dicea_ERROR = 0;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERROR = dicea_ERROR + ((int) j);
                dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) k);
                dicea_ERROR = dicea_ERROR - ((int) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) i);
                dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) j);
                dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) k);
                dicea_ERROR = dicea_ERROR - ((int) k_duplicate);
                dicea_ERROR = dicea_ERROR + ((int) i);
                dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
                if (dicea_ERROR != 0) FaultDetected ();
            }
            dicea_ERROR = dicea_ERROR + ((int) S);
            dicea_ERROR = dicea_ERROR - ((int) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) j);
            dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) i);
            dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
            C (i, j) = S;
            dicea_ERROR = dicea_ERROR + ((int) S);
            dicea_ERROR = dicea_ERROR - ((int) S_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) j);
            dicea_ERROR = dicea_ERROR - ((int) j_duplicate);
            dicea_ERROR = dicea_ERROR + ((int) i);
            dicea_ERROR = dicea_ERROR - ((int) i_duplicate);
            if (dicea_ERROR != 0) FaultDetected ();
        }
}
```

Figura 22 – Código fuente ejemplo con la regla TOI.

5.3.5 Regla TOH

El uso de acumuladores de errores de tipo punto flotante o punto fijo podría generar penalización en las prestaciones por conversión del tipo de la variable para el tipo del acumulador de errores.

Para evitar el coste de las conversiones de tipo hemos diseñado la unión de las dos reglas anteriores, usando entonces dos acumuladores de error: uno de tipo punto flotante y otro de tipo punto fijo.

El acrónimo TOH de esta regla proviene de *Testing Once Hybrid* (testando solamente una vez con tipo hibrido).

Para transformar un código fuente usando la regla TOH hay que seguir los pasos que siguen:

1. Declarar en la aplicación los acumuladores de error:
    a. Una variable global de tipo *double*;
    b. Una variable global de tipo *int*;
2. Declarar una función *FaultDetected* que muestra en la salida patrón de errores (*stderr*) un mensaje avisando de la detección de un fallo y hace una parada segura de la aplicación;
3. Para cada variable local declarada, crear una copia;
4. Para cada atribución hecha a una variable local previamente declarada, repetir la atribución a su copia;
5. Para toda verificación hecha en una variable local previamente declarada, crear una copia de la verificación usando las copias de las variables usadas;
6. Para cada uso de una variable local previamente declarada que aún no tenga sido testeada en el seguimiento de código actual, añadir antes del uso dos declaraciones:
    a. Una que incrementa el acumulador de errores del mismo tipo de la variable con el contenido de la variable original;
    b. Otra que decrementa el acumulador de errores del mismo tipo de la variable con el contenido de la copia de la variable usada;
7. Al final de un seguimiento código, si han tocado algún de los acumuladores de errores con el ítem 6 en el seguimiento actual:
    a. Insertar nuevamente el incremento y el decremento de todas las variables que tengan sido testeadas en el seguimiento de código actual;
    b. Insertar una verificación en los acumuladores de errores tocados.

La Figura 23 presenta el código fuente de la función de multiplicación de matrices presentada anteriormente transformada con la regla TOH.

```
static int dicea_ERRORi = 0;
static double dicea_ERRORd = 0.0f;

static void FaultDetected ()
{
    fprintf (stderr, "Fault Detected.\n");
    exit (-1);
}

void mult ()
{
    int i, j, k;
    int i_duplicate, j_duplicate, k_duplicate;
    double S;
    double S_duplicate;
    for (i = 0, i_duplicate = 0; i < matrixSize, i_duplicate < matrixSize; i++, i_duplicate++)
        for (j = 0, j_duplicate = 0; j < matrixSize, j_duplicate < matrixSize; j++, j_duplicate++)
        {
            S = 0;
            S_duplicate = 0;
            for (k = 0, k_duplicate = 0; k < matrixSize, k_duplicate < matrixSize; k++, k_duplicate++)
            {
                dicea_ERRORi = dicea_ERRORi + ((int) j);
                dicea_ERRORi = dicea_ERRORi - ((int) j_duplicate);
                dicea_ERRORi = dicea_ERRORi + ((int) k);
                dicea_ERRORi = dicea_ERRORi - ((int) k_duplicate);
                dicea_ERRORi = dicea_ERRORi + ((int) i);
                dicea_ERRORi = dicea_ERRORi - ((int) i_duplicate);
                S += A (i, k) * B (k, j);
                S_duplicate += A (i_duplicate, k_duplicate) * B (k_duplicate, j_duplicate);
                dicea_ERRORi = dicea_ERRORi + ((int) j);
                dicea_ERRORi = dicea_ERRORi - ((int) j_duplicate);
                dicea_ERRORi = dicea_ERRORi + ((int) k);
                dicea_ERRORi = dicea_ERRORi - ((int) k_duplicate);
                dicea_ERRORi = dicea_ERRORi + ((int) i);
                dicea_ERRORi = dicea_ERRORi - ((int) i_duplicate);
                if (dicea_ERRORi != 0) FaultDetected ();
            }
            dicea_ERRORd = dicea_ERRORd + ((double) S);
            dicea_ERRORd = dicea_ERRORd - ((double) S_duplicate);
            dicea_ERRORi = dicea_ERRORi + ((int) j);
            dicea_ERRORi = dicea_ERRORi - ((int) j_duplicate);
            dicea_ERRORi = dicea_ERRORi + ((int) i);
            dicea_ERRORi = dicea_ERRORi - ((int) i_duplicate);
            C (i, j) = S;
            dicea_ERRORd = dicea_ERRORd + ((double) S);
            dicea_ERRORd = dicea_ERRORd - ((double) S_duplicate);
            dicea_ERRORi = dicea_ERRORi + ((int) j);
            dicea_ERRORi = dicea_ERRORi - ((int) j_duplicate);
            dicea_ERRORi = dicea_ERRORi + ((int) i);
            dicea_ERRORi = dicea_ERRORi - ((int) i_duplicate);
            if (dicea_ERRORi != 0) FaultDetected ();
            if (dicea_ERRORd != 0) FaultDetected ();
        }
}
```

Figura 23 – Código fuente ejemplo con la regla TOH.

5.4 Automatizando la transformación del código fuente

Hemos desarrollado un entorno compuesto por un conjunto de herramientas y scripts para el sistema operativo Linux para facilitar el trabajo de transformación del código fuente para mejorar la robustez de las aplicaciones. Una visión general de cómo funciona el entorno está en la Figura 24.

Empezamos por usar la herramienta txl [**23**] para transformar un código fuente escrito en lenguaje de programación C en una estructura XML basada en la gramática del lenguaje. La herramienta txl usa el lenguaje TXL, también presentado en [**23**], diseñado específicamente para manipulación y experimentación con lenguajes de programación con recursos que permiten trabajar con transformación de código fuente para código fuente.

Creamos un *framework* que nos permite trabajar haciendo cambios en la estructura XML generada y que también genera un nuevo código fuente basado en la estructura XML.

A ese *framework* llamamos SCOTTION, que proviene de *Source COde TransformaTION*.

Cada una de las reglas de transformación es creada como un programa independiente que usa el mismo *framework* de trasformación del XML.

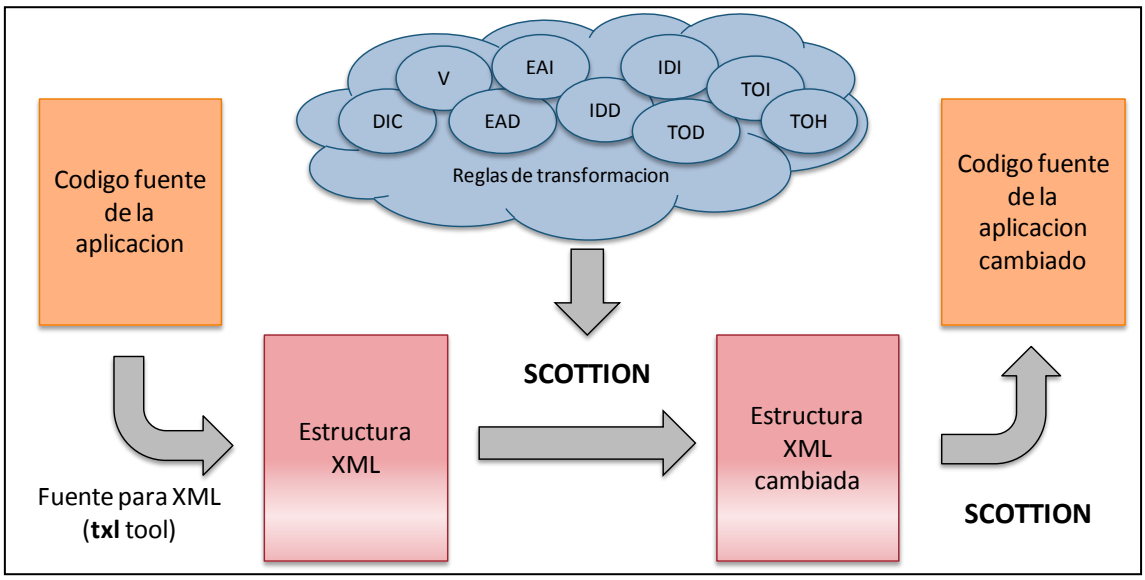

Figura 24 – Diagrama de la automatización del cambio del código fuente.

Cuando llamamos la herramienta SCOTTION necesitamos informar:

1. El fichero con el código fuente de la aplicación que se quiere transformar;
2. La regla de transformación a ser usada;
3. Las funciones del código fuente de serán transformadas.

La herramienta entonces verifica si el fichero con el código fuente existe y después verifica si la regla informada existe. No habiendo errores, la herramienta llama el programa de la transformación informada pasando como parámetros el fichero con el código fuente e las funciones que si quiere transformar. El resultado de la ejecución (el código fuente transformado) es generado si ningún problema es encontrado durante las transformaciones, o sea, si todas las funciones informadas existen y si ningún problema es encontrado en el código fuente original en respecto a la gramática del lenguaje C.

Capítulo 6 Evaluación experimental

6.1 Introducción

En nuestra evaluación experimental, hemos caracterizado la robustez de tres aplicaciones del *NAS Parallel Benchmark* (NPB): *Conjugate Gradient* (CG), *Fast Fourier Transformation* (FT) y *Scalar Pentadiagonal* (SP).

Hemos elegido el NPB por su importancia en estudios de computación de altas prestaciones. Sus aplicaciones hacen cómputos intensivos bastante comunes en problemas reales tratados en los centros de supercomputación. Además, en su versión 2.3 para OpenMP, el NPB dispone de todas sus aplicaciones con código fuente en lenguaje C, lo que indispensable para nuestra herramienta de transformación automatizada de código fuente. Por cuenta de la cantidad de experimentos que teníamos que hacer, hemos elegido el tamaño del problema de las aplicaciones de forma que la versión sin cambios de la aplicación no tardase más que 30 segundos. Así, quedamos con la CG de tamaño A, FT también de tamaño A y SP de tamaño W.

Tabla 4 – Versiones de las aplicaciones experimentadas.

Versión	Descripción
STD	Código fuente original, sin cambios.
DIC	Código fuente transformado con la regla básica de la literatura.
V	Código fuente transformado con la regla básica de la literatura y variables volátiles.
EAD	Código fuente transformado con la regla básica de la literatura (DIC) y acumulador de errores de tipo punto flotante.
EAI	Código fuente transformado con la regla básica de la literatura (DIC) y acumulador de errores de tipo punto fijo.
IDD	Código fuente transformado con la regla EAD pero usando incremento y decremento.
IDI	Código fuente transformado con la regla EAI pero usando incremento y decremento.
TOD	Código fuente transformado con la regla IDD pero reduciendo la cantidad de testes.
TOI	Código fuente transformado con la regla IDI pero reduciendo la cantidad de testes.
TOH	Código fuente transformado con la regla que reduce la cantidad de testes y con acumuladores de errores de punto flotante y fijo.

Todas las tres aplicaciones elegidas tuvieran sus códigos fuentes transformados usando todas las reglas de transformación propuestas en este trabajo, además del caso base recomendado por la literatura, con nuestra herramienta de transformación automatizada de código fuente. De las tres, solamente la FT necesitó de edición manual adicional para sacar errores que le impedían de

compilar correctamente por usar un tipo complejo definido en la aplicación en algunos de los cómputos intermediarios.

En el total, para cada aplicación elegida, quedamos con diez versiones de su código fuente, conforme la Tabla 4.

Cada una de las diez versiones de las tres aplicaciones ha sido compilada usando el compilador C de Intel en su versión 11.1 con tres configuraciones distintas de optimización, conforme la Tabla 5.

Tabla 5 – Optimizaciones del compilador utilizadas.

Optimización	Descripción
O1	Optimización moderada con el objetivo de reducir el tamaño del binario y el tiempo de ejecución de la aplicación, pero sin usar las técnicas que pueden empeorar el tiempo de compilación.
O2	Optimización fuerte que objetiva mejorar el tiempo de ejecución de la aplicación, incluso usando las técnicas que hacen empeorar el tiempo de compilación. También no tiene el compromiso con la reducción del tamaño del binario.
O3	Optimización más agresiva que hay de las opciones patrón.

Con un total de 90 configuraciones distintas (3 aplicaciones × 10 versiones × 3 optimizaciones), hicimos la caracterización de la robustez de cada una de las versiones usando nuestro entorno de caracterización a través de inyección de fallos, ejecutando cada configuración 500 veces (un total de 45 mil ejecuciones).

Las ejecuciones de la caracterización fueran hechas de forma distribuida en los ocho nodos del *cluster* AOHyper del Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO). Cada nodo del AOHyper tiene un procesador AMD Athlon X2 de 64 bits y 2GBytes de memoria RAM. El sistema operativo usado en los nodos del *cluster* AOHyper es el Ubuntu 8.04-LTS para procesadores de 64 bits.

6.2 Eligiendo que transformar

Usando una herramienta de generación de perfil estadístico con amuestras (gprof) de una aplicación para determinar, basado en el código fuente de la aplicación, cuales son las funciones que por más tiempo quedan trabajando cuando la aplicación es ejecutada, hemos elegido la de cada una de las aplicaciones experimentadas la función que más tiempo quedaba trabajando para aplicar la transformación en el código fuente añadiendo la redundancia para detectar los fallos transitorios.

6.2.1 Transformación de la aplicación CG

El resultado de la generación del perfil de la aplicación CG resultó que la función **conj_grad** estaba trabajando en 96,4% de las amuestras hechas por el perfilador.

La función **conj_grad** contiene 11 variables locales: 6 de punto flotante y 5 de punto fijo. De esta forma, no sabemos de antemano cual de las transformaciones que hemos propuesto va mejor (si las con acumulador de errores de tipo punto flotante o de tipo punto fijo), pero creemos que en este caso la transformación con el acumulador de errores hibrido (TOH) tendrá ventaja sobre las demás.

6.2.2 Transformación de la aplicación FT

El resultado de la generación del perfil de la aplicación FT resultó que la función **fftz2** estaba trabajando en 43,2% de las amuestras hechas por el perfilador.

La función **fftz2** contiene 12 variables locales, todas de punto fijo. De esta forma, creemos que las transformaciones con el acumulador de errores de tipo punto fijo tendrán ventaja sobre las demás.

6.2.3 Transformación de la aplicación SP

El resultado de la generación del perfil de la aplicación SP resultó que la función **compute_rhs** estaba trabajando en 32,34% de las amuestras hechas por el perfilador.

La función **compute_rhs** contiene 15 variables locales: 4 de tipo punto fijo y 11 de tipo punto flotante. De esta forma, así como con la aplicación CG, no sabemos de antemano cual de las transformaciones que hemos propuesto va mejor (si las con acumulador de errores de tipo punto flotante o de tipo punto fijo), pero también creemos que en este caso la transformación con el acumulador de errores hibrido (TOH) tendrá ventaja sobre las demás.

6.3 Análisis general de los resultados

En esta sesión, analizaremos los resultados obtenidos con las tres aplicaciones elegidas, empezando por el análisis con la optimización más agresiva del compilador, siguiendo por la optimización fuerte y terminando con la optimización moderada.

Al final de la sesión, haremos una comparación general de la robustez de nuestras propuestas y también de cómo hemos afectado el equilibrio entre la robustez obtenida y la sobrecarga añadida a través de la comparación de los MWTF obtenidos.

6.3.1 Experimentación con la optimización más agresiva (O3)

6.3.1.1 Aplicación CG con optimización O3

Analizando el grafico de la Figura 25, notamos que casi todas las transformaciones que hemos propuesto mejoran la transformación DIC, con la cual nos comparamos.

La ultima regla de transformación que hemos propuesto (TOH) ha logrado el mejor resultado de robustez, haciendo con que la cantidad de SDC rebajase en cerca de 28,4% (de 45,8% con el DIC hasta 32,8% con el TOH). Comparando nuestro mejor resultado con la aplicación sin redundancia notamos una reducción de 48,9% (de 64,2% en la configuración STD hasta 32,8% con el TOH).

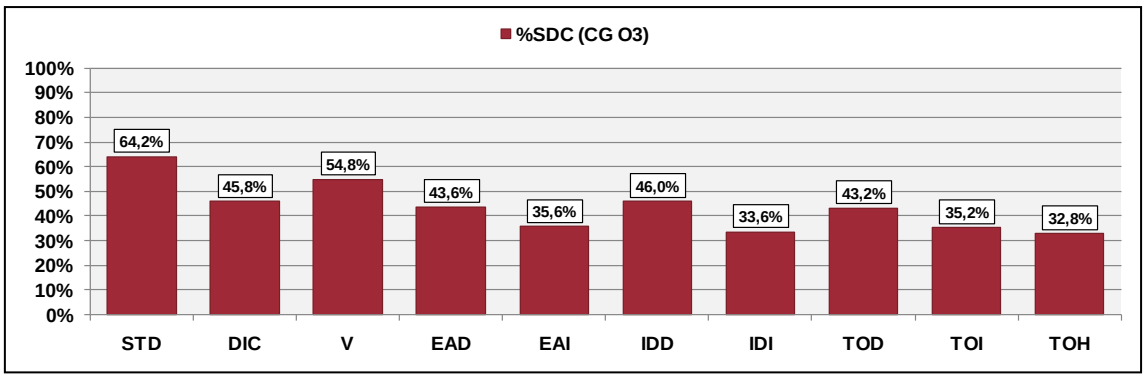

Figura 25 – Robustez de la aplicación CG con optimización O3.

Analizando el tiempo de ejecución normalizado de la aplicación CG con la optimización O3 en la Figura 26 notamos la grande sobrecarga añadida por la regla V, llevando la aplicación a tardar cerca de 2,82 veces más que la versión sin detección de fallos (STD).

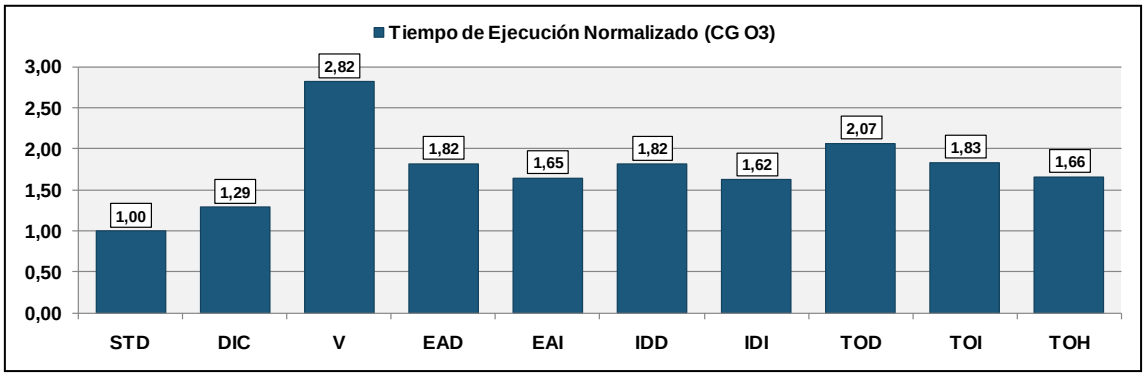

Figura 26 – Tiempo de ejecución normalizado de la aplicación CG con optimización O3.

La sobrecarga añadida por la regla DIC es la menor (cerca de 29%) de las reglas de transformación presentadas. Las reglas con el acumulador de errores de tipo punto flotante son las mayores después de la V, llegando con la regla TOD a hacer la aplicación tardar más que el doble de la aplicación sin detección de fallos.

Después de analizar la robustez y la sobrecarga añadida por las reglas de transformación, pasamos a analizar entonces la métrica MWTF, que se presenta graficada de forma normalizada en la Figura 27.

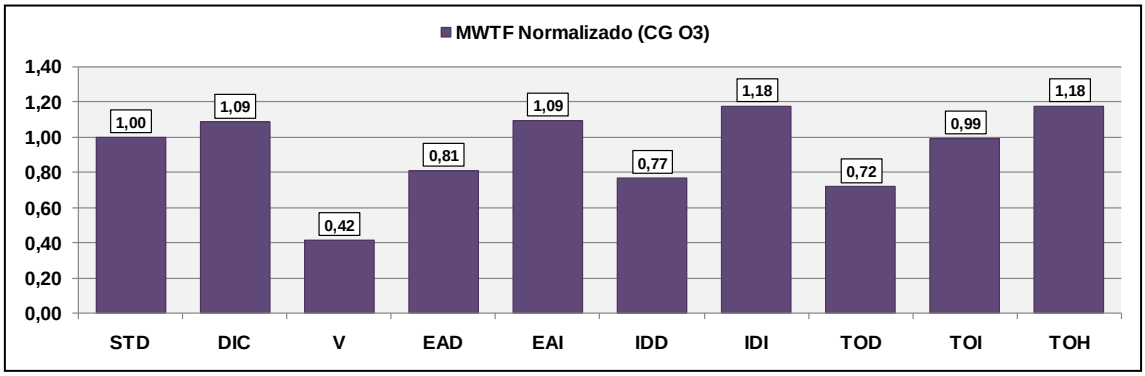

Figura 27 – MWTF normalizado de la aplicación CG con optimización O3.

Notamos a penalización sufrida por la regla V por su sobrecarga, dejando su MWTF como el menor obtenido en este conjunto de experimentos. Notamos también que la regla DIC, que había logrado la menor sobrecarga, mejora el MWTF en relación a la aplicación sin detección de fallos en un 9%.

Dos de las reglas quedaran empatadas con el mejor MWTF de este conjunto de experimentos: las reglas IDI y TOH. Las dos lograran un MWTF 18% superior a la aplicación sin detección de fallos (el doble de la regla base DIC).

6.3.1.2 Aplicación FT con optimización O3

Analizando el grafico de la Figura 28, notamos un comportamiento bastante diferente del presentado anteriormente con la aplicación CG en la Figura 25. Si con la aplicación CG habíamos logrado con que todas las transformaciones que hemos propuesto mejorasen la transformación DIC, con la aplicación FT tuvimos tres reglas de transformación que quedaran en un umbral cercano a la transformación DIC: las reglas EAD, IDD y TOD, todas empleando el acumulador de errores de tipo punto flotante, situación que ya habíamos previsto.

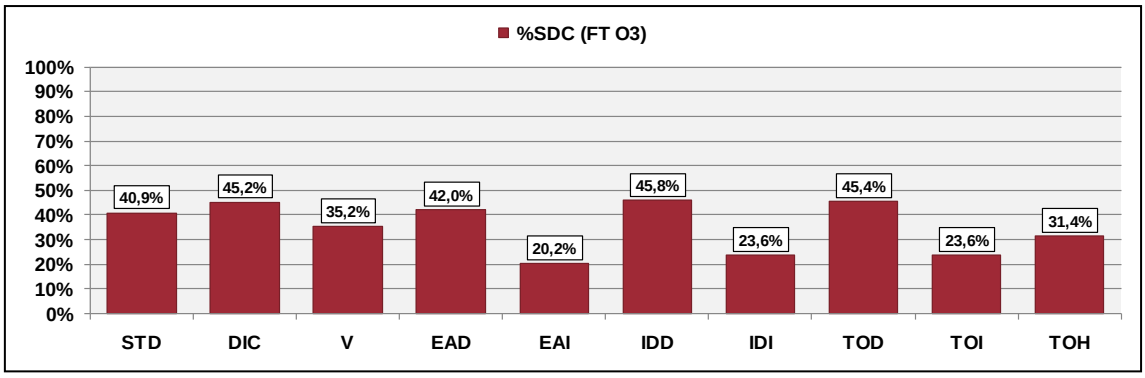

Figura 28 – Robustez de la aplicación FT con optimización O3.

Además, la regla DIC no ha logrado mejorar la robustez de la aplicación, pues la cantidad de SDC ha subido de 40,9% para 45,2% (una peoría de 10,6%).

La primera regla de transformación con acumulador de errores que hemos propuesto (EAI) ha logrado el mejor resultado de robustez, haciendo con que la cantidad de SDC rebajase en cerca de 50,6% (de 40,9% en la configuración STD hasta 20,2% con el EAI).

Analizando el tiempo de ejecución normalizado de la aplicación FT con la optimización O3 en la Figura 29 notamos la grande sobrecarga añadida por la regla V además de la sobrecarga generada por las conversiones de tipo en las reglas de transformación con acumulador de errores de tipo punto flotante.

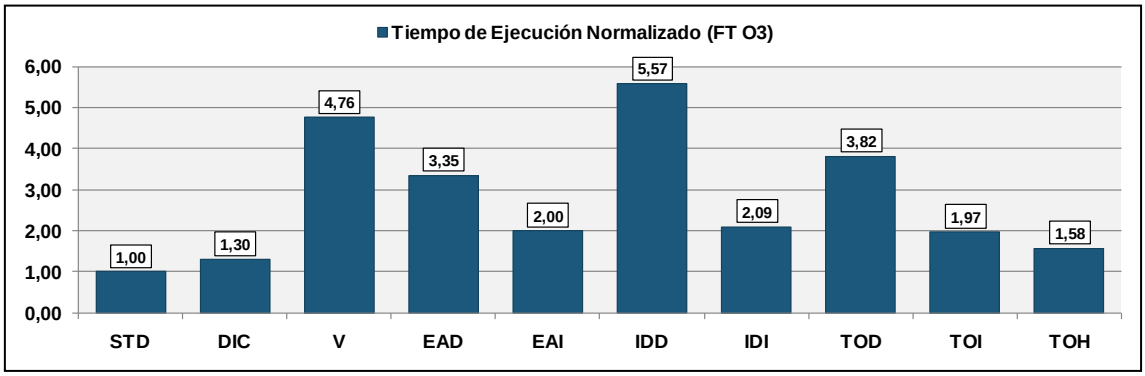

Figura 29 – Tiempo de ejecución normalizado de la aplicación FT con optimización O3.

La sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 30%) de las reglas de transformación presentadas. Las reglas con el acumulador de errores de tipo punto fijo se han mantenido en el umbral del 100% de sobrecarga, y de las reglas que hemos propuesto, la menor sobrecarga hemos logrando con la regla TOH, cerca de 58%.

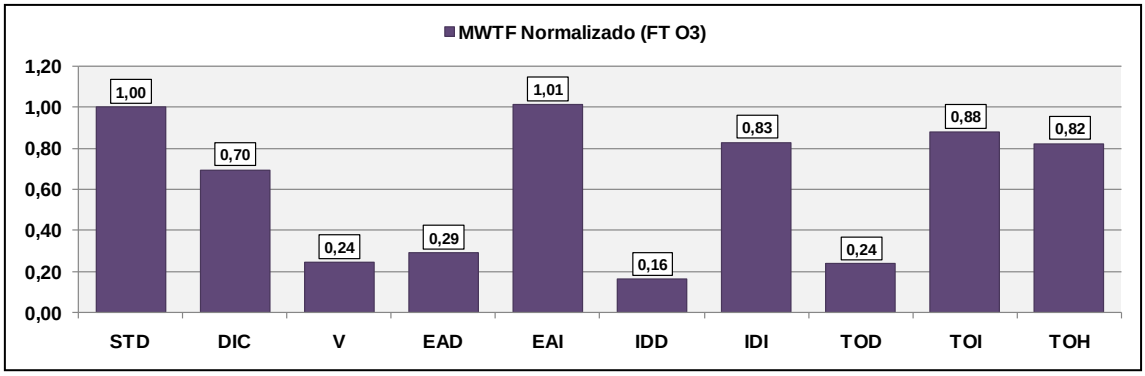

Figura 30 – MWTF normalizado de la aplicación FT con optimización O3.

Analizando entonces la métrica MWTF, que se presenta graficada de forma normalizada en la Figura 30, notamos que la penalización sufrida por la sobrecarga en las regla V, EAD, IDD y TOD dejan sus MWTF por debajo de 30% del MWTF de la aplicación sin detección de fallos.

Notamos también que la regla DIC, que había logrado la menor sobrecarga, todavía no mejora el MWTF en relación a la aplicación sin detección de fallos.

La única regla que llega a mejorar el MWTF en este conjunto de experimentos en relación a la aplicación sin detección de fallos es la EAI, pero la mejoría de cerca de 1% no llega a ser significativa.

6.3.1.3 Aplicación SP con optimización O3

Analizando el grafico de la Figura 31, notamos que por la primera vez la regla V logra la mejor robustez. Además de eso, tuvimos un resultado similar al de la aplicación CG, donde todas las transformaciones que hemos propuesto mejoran la transformación DIC.

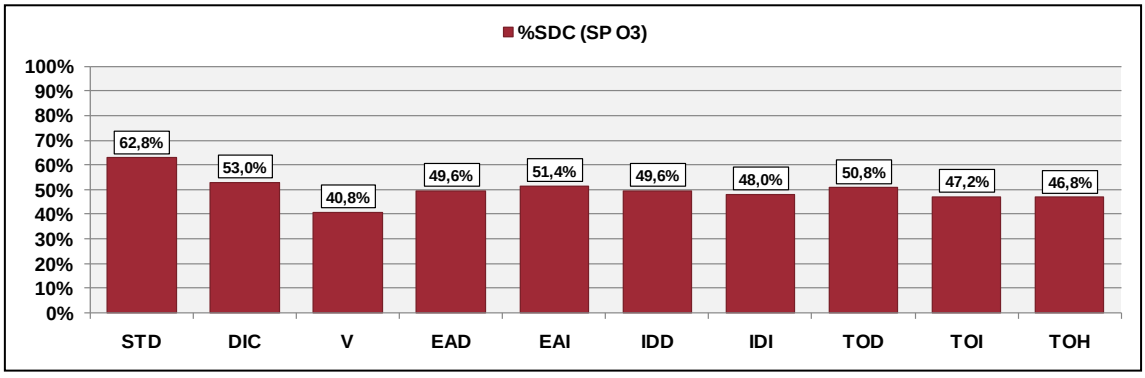

Figura 31 – Robustez de la aplicación SP con optimización O3.

La regla DIC ha logrado una mejora en la robustez de 15,6% y la regla V ha llegado a 35% de mejoría. Afuera la regla V, la regla TOH ha logrado 25,5% de mejoría en la robustez en relación a la aplicación sin detección de fallos (11,7% de mejoría en relación a la regla DIC), algo que ya habíamos previsto por cuenta de que la función en que hemos aplicado las reglas de transformación cuenta con variables de tipo punto fijo y flotante.

Una vez más, en el tiempo de ejecución normalizado de la aplicación SP con la optimización O3 de la Figura 32, notamos la grande sobrecarga añadida por la regla V.

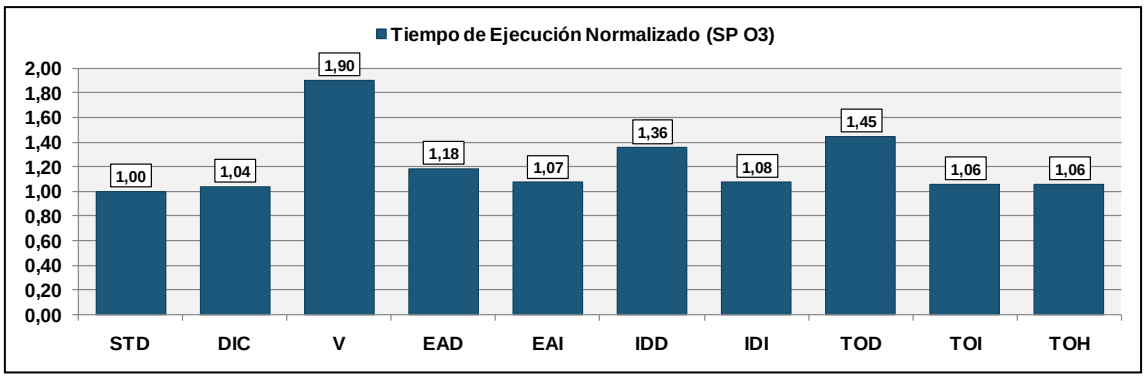

Figura 32 – Tiempo de ejecución normalizado de la aplicación SP con optimización O3.

Una vez más, la sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 4%) de las reglas de transformación presentadas. Las reglas con el acumulador de errores de tipo punto fijo se han mantenido en el umbral del 6% a 8% de sobrecarga, y de las reglas que hemos propuesto, la menor sobrecarga hemos logrando con las reglas TOI y TOH, cerca de 6%.

Analizando la métrica MWTF, que se presenta graficada de forma normalizada en laFigura 33, notamos que la penalización sufrida por la sobrecarga en las regla V, IDD y TOD dejan sus MWTF por debajo del MWTF de la aplicación sin detección de fallos.

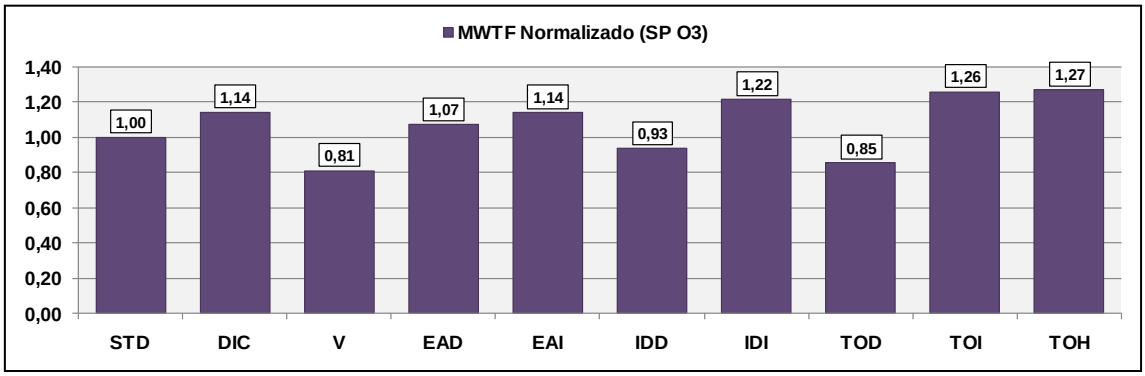

Figura 33 – MWTF normalizado de la aplicación SP con optimización O3.

Podemos notar que la regla DIC, que había logrado la menor sobrecarga, mejora el MWTF en relación a la aplicación sin detección de fallos en un 14%, pero el mejor MWTF de este conjunto de experimentos hemos logrado con la regla TOH, que llegó una mejora de 27% en el MWTF en relación a la aplicación sin detección de fallos (casi el doble de la regla base DIC).

6.3.2 Experimentación con optimización fuerte (O2)

6.3.2.1 Aplicación CG con optimización O2

Analizando el grafico de la Figura 34, igual al que nos ha pasado con la optimización más agresiva, notamos que casi todas las transformaciones que hemos propuesto mejoran la transformación DIC.

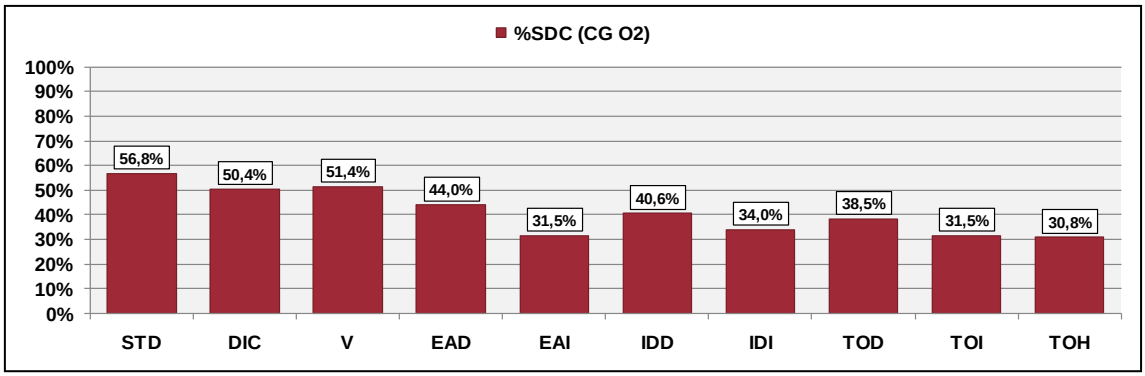

Figura 34 – Robustez de la aplicación CG con optimización O2.

La ultima regla de transformación que hemos propuesto (TOH) sigue con el mejor resultado de robustez, haciendo con que la cantidad de SDC rebajase en cerca de 38,8% (de 50,4% con el DIC hasta 30,8% con el TOH). Comparando nuestro mejor resultado con la aplicación sin redundancia notamos una reducción de 45,7% (de 56,8% en la configuración STD hasta 30,8% con el TOH). Mismo logrando mejorar la robustez en relación a la optimización O3 (de 32,8% para 30,8%), si analizamos la mejoría en relación a la configuración STD que hemos logrado con la configuración TOH ha disminuido de 48,9% para 45,7%.

Analizando el tiempo de ejecución normalizado de la aplicación CG con la optimización O2 en la Figura 35 seguimos notando la grande sobrecarga añadida por la regla V, llevando la aplicación a tardar cerca de 2,71 veces más que la versión sin detección de fallos (STD).

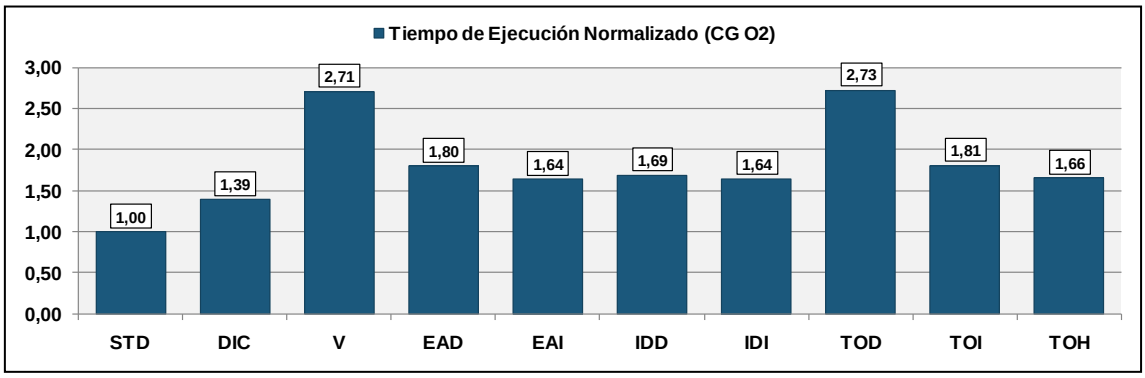

Figura 35 – Tiempo de ejecución normalizado de la aplicación CG con optimización O2.

La sobrecarga añadida por la regla DIC también sigue siendo la menor (cerca de 39%) de las reglas de transformación presentadas, mismo tardando un 10% más en relación a su versión STD que la optimización más agresiva. Las reglas con el acumulador de errores de tipo punto flotante siguen peores que las con acumulador de errores de tipo punto fijo, pero ahora tenemos como peor caso la regla TOD con 2,73 veces el tiempo de la aplicación sin detección de fallos.

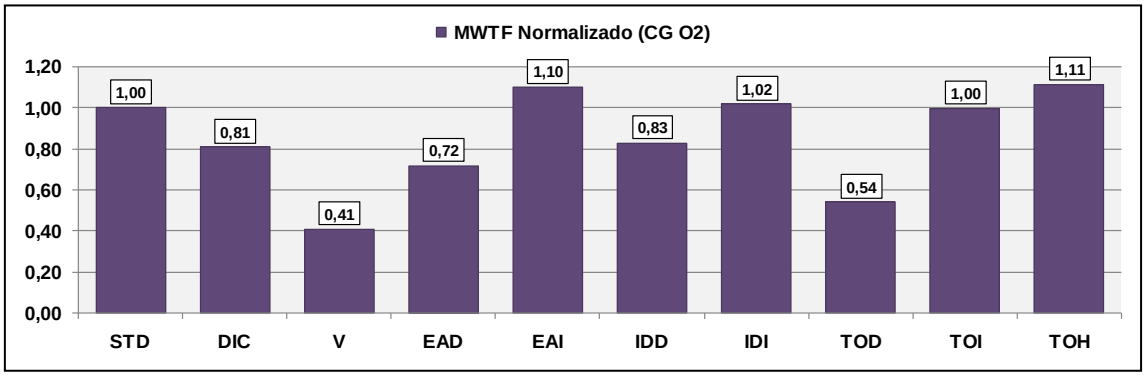

Figura 36 – MWTF normalizado de la aplicación CG con optimización O2.

Notamos en la Figura 36 la penalización sufrida por las reglas V y TOD por sus sobrecargas en el tiempo de ejecución, dejando sus MWTF como los menores obtenidos en este conjunto de

experimentos. Notamos también que la regla DIC, de esta vez ya no logra mejorar el MWTF en relación a la aplicación sin detección de fallos.

Todas las reglas que hemos propuesto con los acumuladores de errores de tipo punto fijo han igualado o mejorado el MWFT en relación a la aplicación sin detección de fallos (EAI, IDI y TOI). Además de estas, la regla TOH logra una vez más el mejor MWTF de este conjunto de experimentos, un 11% superior a la aplicación sin detección de fallos.

6.3.2.2 Aplicación FT con optimización O2

Analizando el grafico de la Figura 37, notamos un comportamiento parecido con el presentado anteriormente con la optimización más agresiva. Sin embargo las tres reglas de transformación que quedaban en un umbral cercano a la transformación DIC, las EAD, IDD y TOD, ahora se presentan resultados peores que la regla DIC, mismo teniendo mejorado sus propios resultados en relación a las versiones compiladas con la optimización más agresiva.

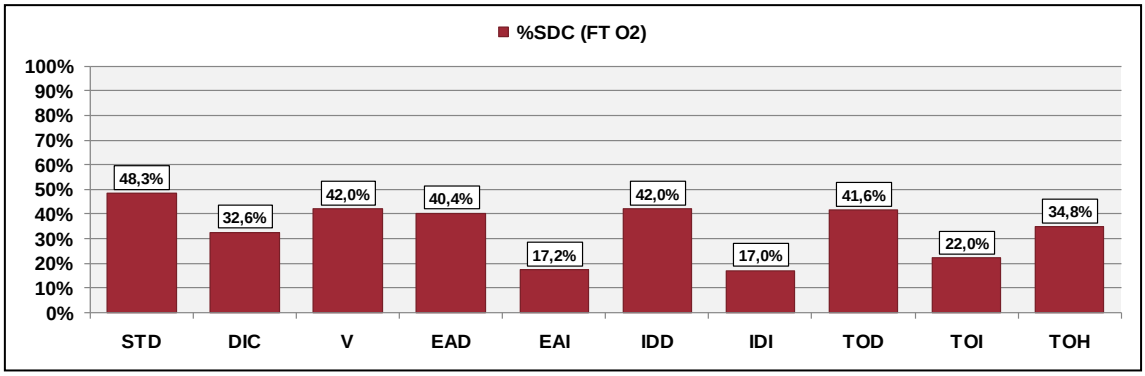

Figura 37 – Robustez de la aplicación FT con optimización O2.

Ahora, la regla DIC ha logrado mejorar la robustez de la aplicación en un 32,5% (de 48,3% en la configuración STD para 32,6%).

La primera regla de transformación con acumulador de errores que hemos propuesto (EAI) ha logrado el segundo mejor resultado de robustez, haciendo con que la cantidad de SDC rebajase en cerca de 64,4% (de 48,3% en la configuración STD hasta 17,2% con el EAI), quedando por detrás del resultado obtenido por la regla IDI que ha rebajado la cantidad de SDC de en relación a la versión STD en un 64,7% (de 48,3% para 17,0%).

Analizando el tiempo de ejecución normalizado de la aplicación FT con la optimización O2 en la Figura 38 seguimos notamos la grande sobrecarga añadida por la regla V además de la sobrecarga generada por las conversiones de tipo en las reglas de transformación con acumulador de errores de tipo punto flotante (EAD, IDD y TOD).

La sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 38%) de las reglas de transformación presentadas y las reglas con el acumulador de errores de tipo punto fijo se han

mantenido en el umbral entre 111% y 123% de sobrecarga. También, igual que nos ha pasado con la optimización más agresiva, la menor sobrecarga hemos logrando fue con la regla TOH, cerca de 68%.

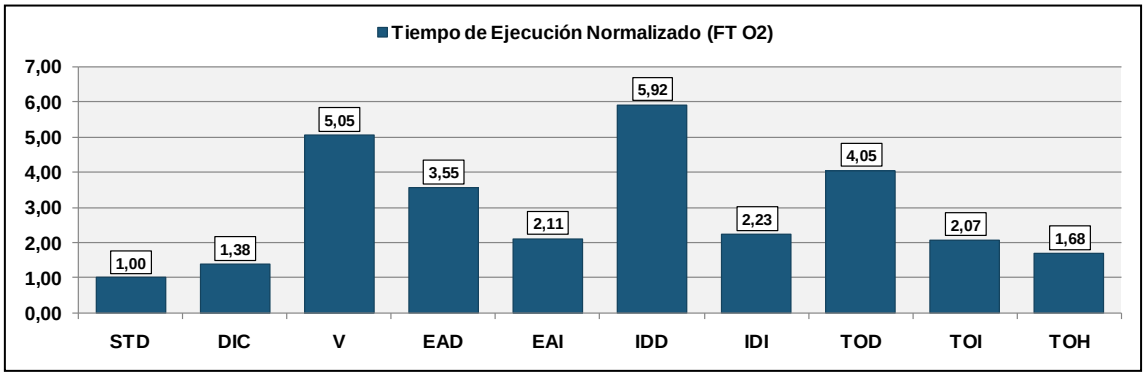

Figura 38 – Tiempo de ejecución normalizado de la aplicación FT con optimización O2.

Analizando entonces la métrica MWTF de la aplicación FT compilada con optimización O2 en la Figura 39, notamos que la penalización sufrida por la sobrecarga en las regla V, EAD, IDD y TOD sigue dejando sus MWTF como los más bajos, de esta vez por debajo de 34% del MWTF de la aplicación sin detección de fallos.

Notamos también que la regla DIC ahora mejora el MWTF en relación a la aplicación sin detección de fallos en un 7%.

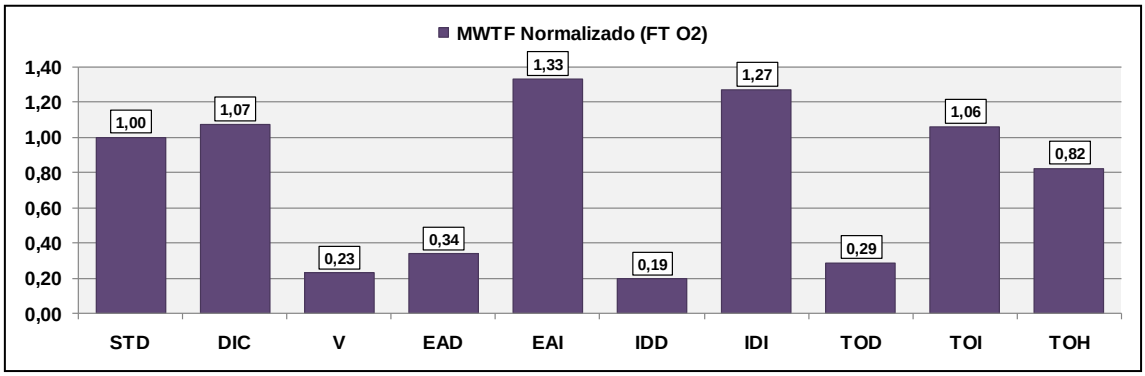

Figura 39 – MWTF normalizado de la aplicación FT con optimización O2.

La mejora en la robustez de las reglas EAI, IDI y TOI hicieron con que todas ellas lograsen un MWTF mejor que la aplicación sin detección de fallos, pero solamente las EAI y IDI han logrado un MWTF mejor que la regla DIC. La mejora lograda, diferente de la lograda en los experimentos con la optimización más agresiva, ahora es bastante significativa, llegando a 33% de mejora en relación a la aplicación sin detección de fallos (un 26% más que la regla DIC).

6.3.2.3 Aplicación SP con optimización O2

Analizando el grafico de la Figura 40, notamos que la regla V sigue logrando la mejor robustez. Además de eso, tuvimos un resultado casi similar al obtenido con la optimización más agresiva,

donde todas las transformaciones que hemos propuesto mejoran la transformación DIC. De esta vez, solamente la regla TOI de las reglas propuestas no ha logrado mejorar la robustez de la regla DIC.

La regla DIC ha logrado una mejora en la robustez de 19% y la regla V ha llegado a 38,9% de mejoría. Afuera la regla V, nuestro mejor resultado hemos obtenido con la regla EAD, que ha logrado 31,5% de mejoría en la robustez en relación a la aplicación sin detección de fallos (15,4% de mejoría en relación a la regla DIC).

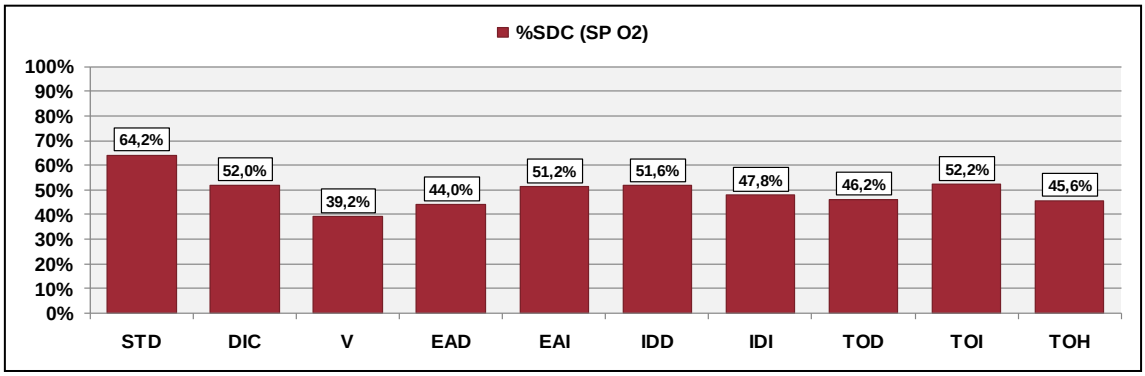

Figura 40 – Robustez de la aplicación SP con optimización O2.

Una vez más, en el tiempo de ejecución normalizado de la aplicación SP con la optimización O2 de la Figura 41, notamos la grande sobrecarga añadida por la regla V.

La sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 5%) de las reglas de transformación presentadas, pero de esta vez un de las reglas que hemos propuesto ha igualado esta sobrecarga: la TOI. Además, las reglas EAI, IDI y TOH ya llegan muy cerca de la sobrecarga mínima obtenida, se mantenido en el umbral del 6% a 7% de sobrecarga.

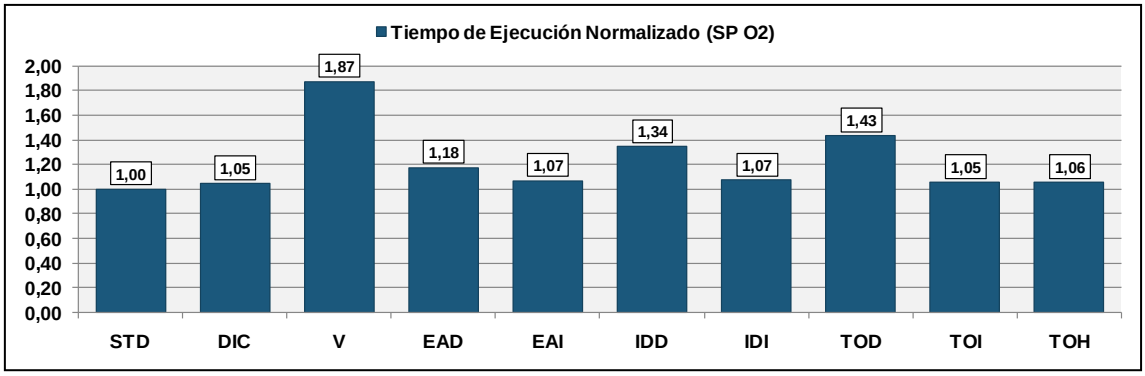

Figura 41 – Tiempo de ejecución normalizado de la aplicación SP con optimización O2.

Analizando la métrica MWTF, que se presenta graficada de forma normalizada en la Figura 42, seguimos notando la penalización sufrida por la sobrecarga en las reglas V, IDD y TOD dejan sus MWTF por debajo del MWTF de la aplicación sin detección de fallos.

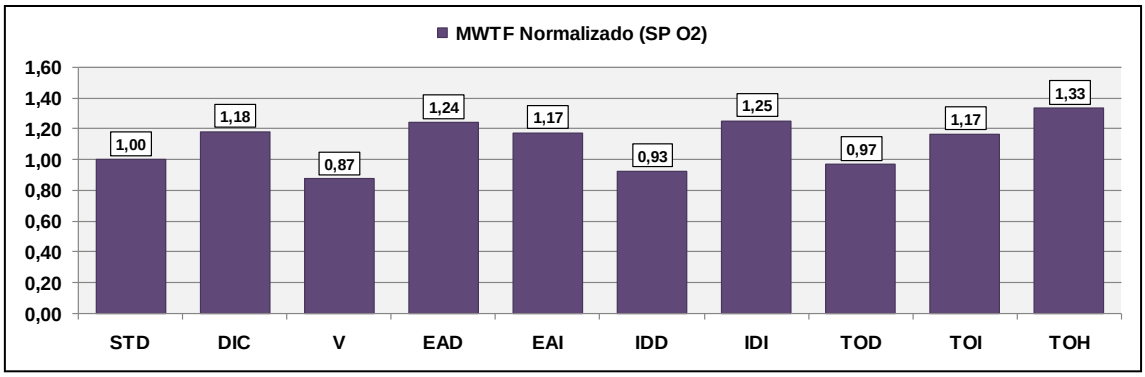

Figura 42 – MWTF normalizado de la aplicación SP con optimización O2.

La regla DIC, que había logrado la menor sobrecarga, mejora el MWTF en relación a la aplicación sin detección de fallos en un 18%, pero el mejor MWTF de este conjunto de experimentos hemos logrado con la misma regla del experimento anterior con la aplicación SP, la regla TOH, que llegó una mejora de 33% en el MWTF en relación a la aplicación sin detección de fallos.

6.3.3 Experimentación con optimización moderada (O1)

6.3.3.1 Aplicación CG con optimización O1

Analizando el grafico de la Figura 43, igual al que nos ha pasado con las optimizaciones anteriores, notamos que casi todas las transformaciones que hemos propuesto mejoran la transformación DIC. Solamente la transformación V obtuve un resultado peor que la transformación DIC.

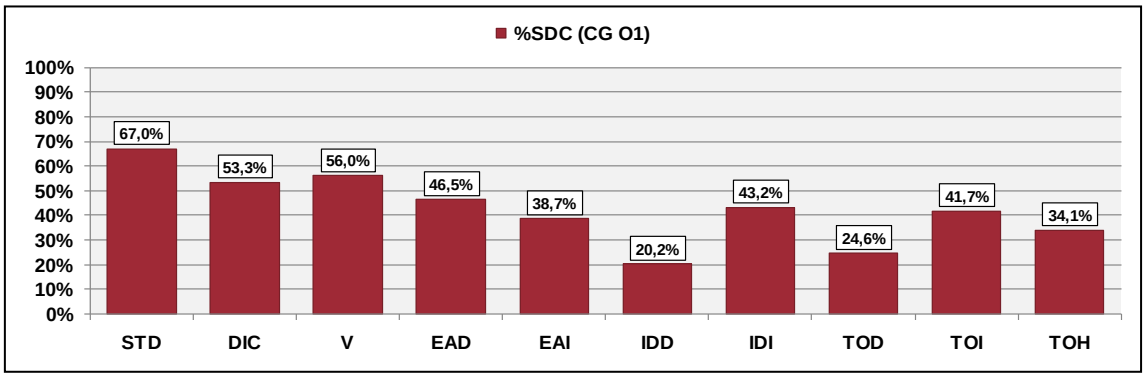

Figura 43 – Robustez de la aplicación CG con optimización O1.

La ultima regla de transformación que hemos propuesto (TOH) no logra más obtener el mejor resultado de robustez. La transformación con la regla IDD ha logrado el mejor resultado de este conjunto de experimentos, haciendo con que la cantidad de SDC rebajase en cerca de 62,1% (de 53,3% con la DIC hasta 20,2% con la IDD). Comparando nuestro mejor resultado con la aplicación sin redundancia notamos una reducción de 69,9% (de 67,0% en la configuración STD hasta 20,2% con la regla IDD). En este conjunto también hemos logrado mejorar tanto la

robustez en general en relación a las optimizaciones anteriores (de 32,8% con O3 y 30,8% con O2 para 20,2% con la optimización O1) cuanto la mejoría de la robustez en relación a la configuración STD, que pasó de 48,9% con la optimización O3 para 45,7% con la optimización O2 y ha llegado a 69,9% con la optimización O1.

Por otro lado, analizando el tiempo de ejecución normalizado de la aplicación CG con la optimización O1 en la Figura 44 notamos, además de la grande sobrecarga añadida por la regla V, llevando la aplicación a tardar cerca de 2,61 veces más que la versión sin detección de fallos (STD), que las dos reglas que han logrado los mejores resultados de la robustez (IDD y TOD) han quedado con tiempo de ejecución mayor que el doble del tiempo de ejecución de la aplicación sin detección de fallos, llegando a 3,18.

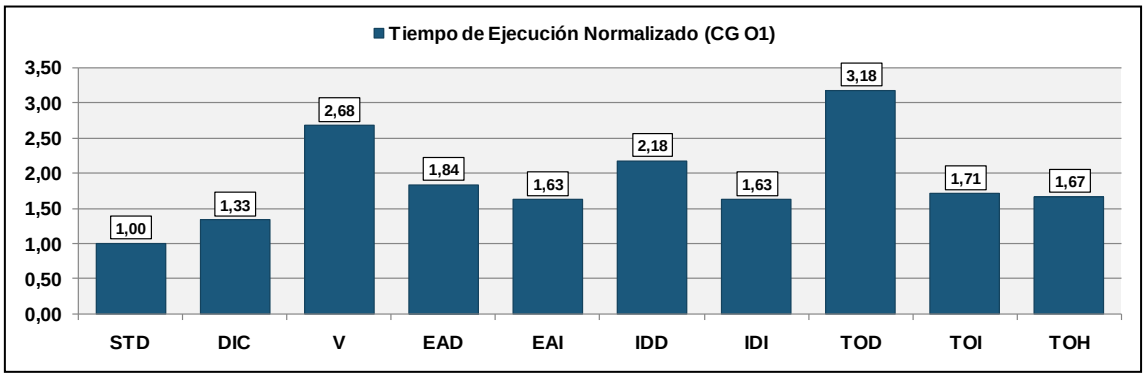

Figura 44 – Tiempo de ejecución normalizado de la aplicación CG con optimización O1.

La sobrecarga añadida por la regla DIC también sigue siendo la menor (cerca de 33%) de las reglas de transformación presentadas. Las reglas con el acumulador de errores de tipo punto flotante siguen peores que las con acumulador de errores de tipo punto fijo y tal como hemos notado en los resultados con optimización O2, tenemos como peor caso la regla TOD ahora con 3,18 veces el tiempo de la aplicación sin detección de fallos.

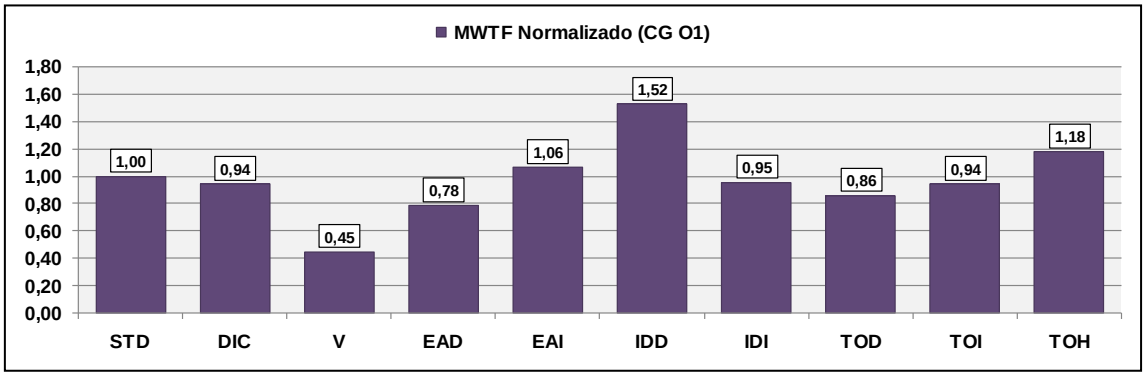

Figura 45 – MWTF normalizado de la aplicación CG con optimización O1.

Notamos en la Figura 45 la penalización sufrida por las reglas V y TOD por sus sobrecargas en el tiempo de ejecución, dejando sus MWTF como los menores obtenidos en este conjunto de

experimentos. Notamos también que la regla DIC sigue sin mejorar el MWTF en relación a la aplicación sin detección de fallos.

La regla TOH de esta vez no ha logrado el mejor MWTF de este conjunto de experimentos, mismo quedando un resultado un 18% superior a la aplicación sin detección de fallos (habíamos logrado 11% con la optimización O2). La reducción en la cantidad de SDC obtenida por la regla IDD ha sido fundamental para la obtención del mejor MWTF de este conjunto de experimentos que, mismo con un tiempo de ejecución relativamente alto, ha mejorado el MWTF en relación a la aplicación sin detección de fallos en un 52%.

6.3.3.2 Aplicación FT con optimización O1

Analizando el grafico de la Figura 46, notamos que el comportamiento sigue parecido con el presentado en las optimizaciones anteriores para esta aplicación. Sin embargo las tres reglas de transformación que ya habían presentado resultados peores que la regla DIC, las EAD, IDD y TOD, siguen con la misma tendencia.

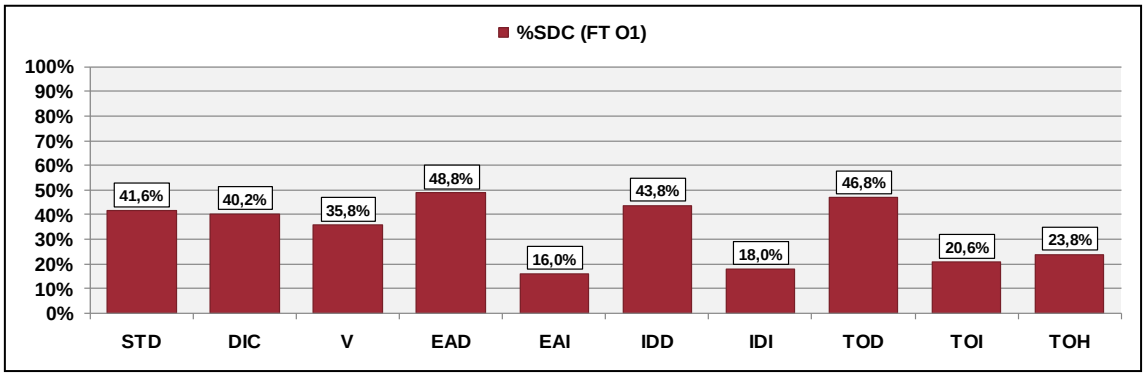

Figura 46 – Robustez de la aplicación FT con optimización O1.

La regla DIC casi no ha logrado mejorar la robustez de la aplicación rebajando de 41,6% en la configuración STD para 40,2%, una mejora de solamente 3,4%.

Igual que la configuración con la optimización más agresiva, la primera regla de transformación con acumulador de errores que hemos propuesto (EAI) ha logrado el mejor resultado de robustez, haciendo con que la cantidad de SDC rebajase en cerca de 61,5% (de 41,6% en la configuración STD hasta 16,0% con el EAI). Este resultado ha sido un poco mejor al obtenido por la regla IDI, que ha rebajado la cantidad de STD en un 56,7%.

Analizando el tiempo de ejecución normalizado de la aplicación FT con la optimización O1 en la Figura 47 notamos que las reglas de transformación con acumulador de errores de tipo punto flotante (EAD, IDD y TOD) tienen una sobrecarga bastante elevada, llegando a 660% del tiempo de ejecución de la aplicación sin detección de fallos.

La sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 26%) de las reglas de transformación presentadas y las reglas con el acumulador de errores de tipo punto fijo se han mantenido en el umbral entre 84% y 119% de sobrecarga. Ahora, diferentemente de lo que nos ha pasado con las optimizaciones anteriores, la menor sobrecarga hemos logrando fue con la regla TOI, cerca de 84%.

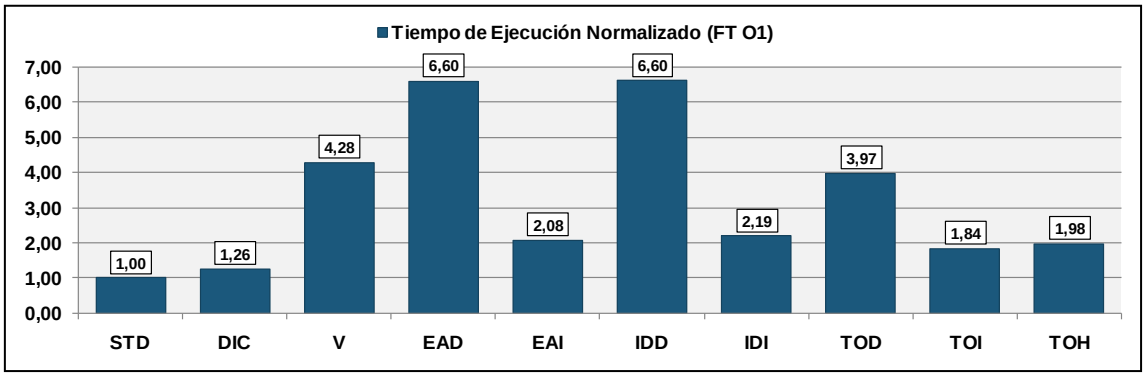

Figura 47 – Tiempo de ejecución normalizado de la aplicación FT con optimización O1.

Analizando entonces la métrica MWTF de la aplicación FT compilada con optimización O1 en la Figura 48, notamos que la penalización sufrida por la sobrecarga en las regla V, EAD, IDD y TOD sigue dejando sus MWTF como los más bajos, de esta vez por debajo de 27% del MWTF de la aplicación sin detección de fallos.

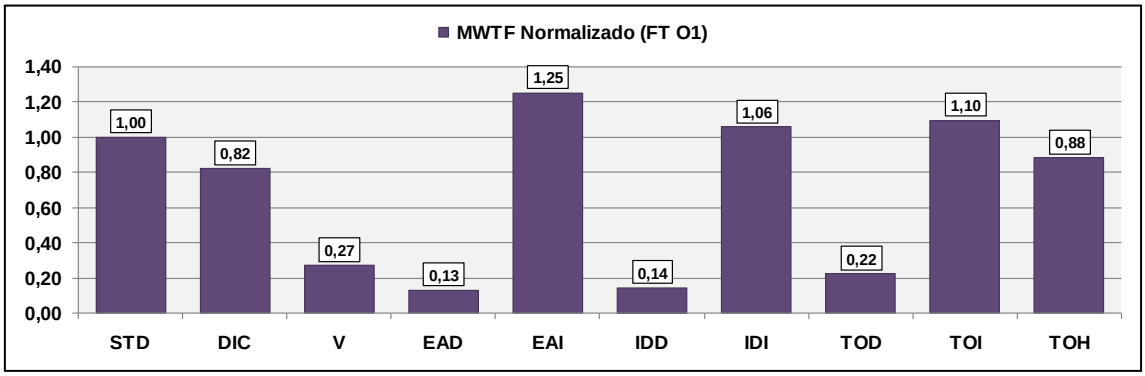

Figura 48 – MWTF normalizado de la aplicación FT con optimización O1.

Notamos también que la regla DIC pasa una vez más a no mejorar el MWTF en relación a la aplicación sin detección de fallos.

La mejora en la robustez de las reglas EAI, IDI y TOI hizo con que solamente ellas lograsen un MWTF mejor que la aplicación sin detección de fallos. La mejora lograda llegó a 25% de mejora en relación a la aplicación sin detección de fallos con la regla EAI (un 25% más que la versión STD).

6.3.3.3 Aplicación SP con optimización O1

Analizando el grafico de la Figura 49, notamos que la regla V sigue logrando la mejor robustez, el mismo que ha pasado con las optimizaciones anteriores. Además de eso, seguimos con todas las transformaciones que hemos propuesto mejorando la transformación DIC.

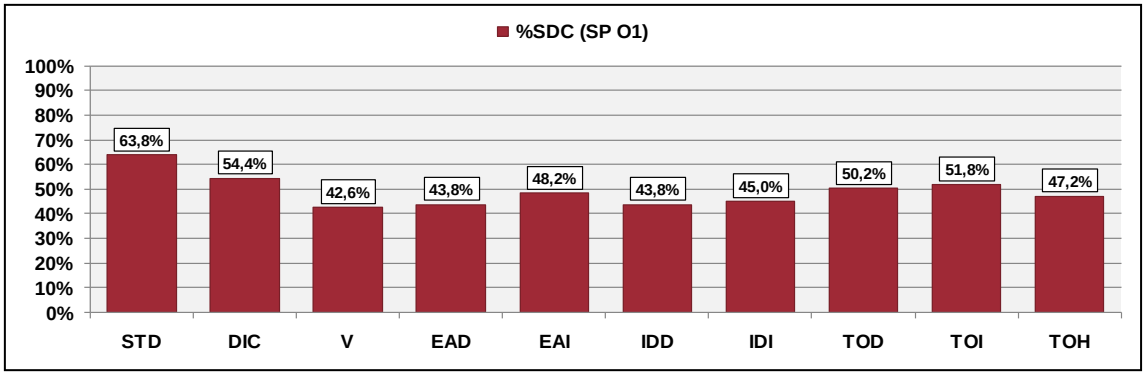

Figura 49 – Robustez de la aplicación SP con optimización O1.

La regla DIC ha logrado una mejora en la robustez de 14,7% y la regla V ha llegado a 33,2% de mejoría. Afuera la regla V, nuestro mejor resultado hemos obtenido con las reglas EAD y IDD, que han logrado 31,3% de mejoría en la robustez en relación a la aplicación sin detección de fallos (19,5% de mejoría en relación a la regla DIC).

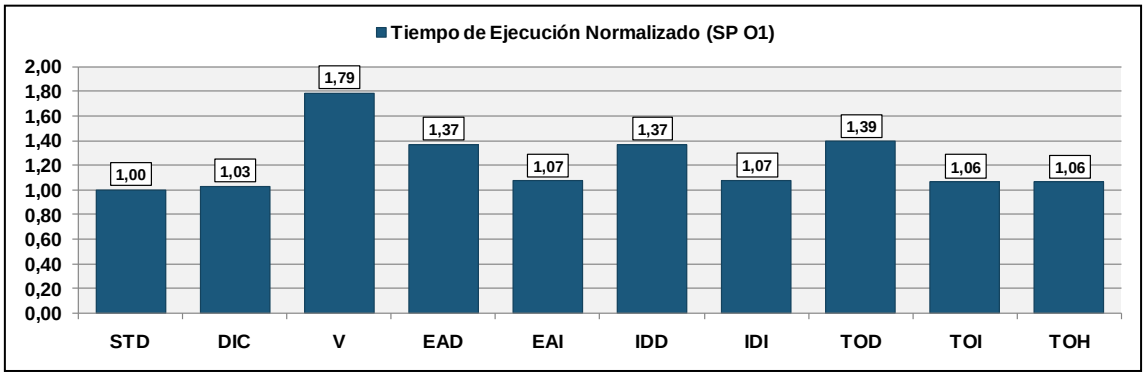

Figura 50 – Tiempo de ejecución normalizado de la aplicación SP con optimización O1.

Una vez más, en el tiempo de ejecución normalizado de la aplicación SP con la optimización O2 de la Figura 50, notamos la grande sobrecarga añadida por la regla V.

La sobrecarga añadida por la regla DIC sigue siendo la menor (cerca de 3%) de las reglas de transformación presentadas y las reglas EAI, IDI, TOI y TOH ya llegan muy cerca de la sobrecarga mínima obtenida, se mantenido en el umbral del 6% a 7% de sobrecarga.

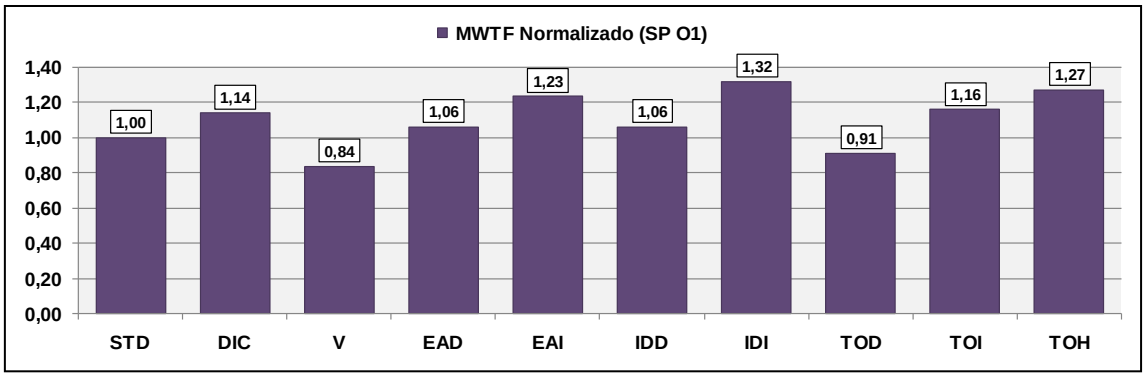

Figura 51 – MWTF normalizado de la aplicación SP con optimización O1.

Analizando la métrica MWTF, que se presenta graficada de forma normalizada en la Figura 51, seguimos notando la penalización sufrida por la sobrecarga en las reglas V y TOD que dejan sus MWTF por debajo del MWTF de la aplicación sin detección de fallos.

La regla DIC, que había logrado la menor sobrecarga, mejora el MWTF en relación a la aplicación sin detección de fallos en un 14% y el mejor MWTF de este conjunto de experimentos hemos logrado con la regla IDI, que llegó una mejora de 32% en el MWTF en relación a la aplicación sin detección de fallos.

6.3.4 Análisis general de la robustez

Comparando la cantidad de resultados SDC obtenidos en los experimentos, hemos notado que en todas las aplicaciones que hemos probado hemos notado que algunas de nuestras propuestas usando un acumulador de errores han logrado mejores resultados de robustez en relación a la regla DIC.

En la aplicación CG, como podemos observar en la Figura 52, hemos logrado un 69,9% de mejoría en la robustez en relación a la aplicación sin detección de fallos en nuestro mejor resultado con la regla IDD y optimización moderada.

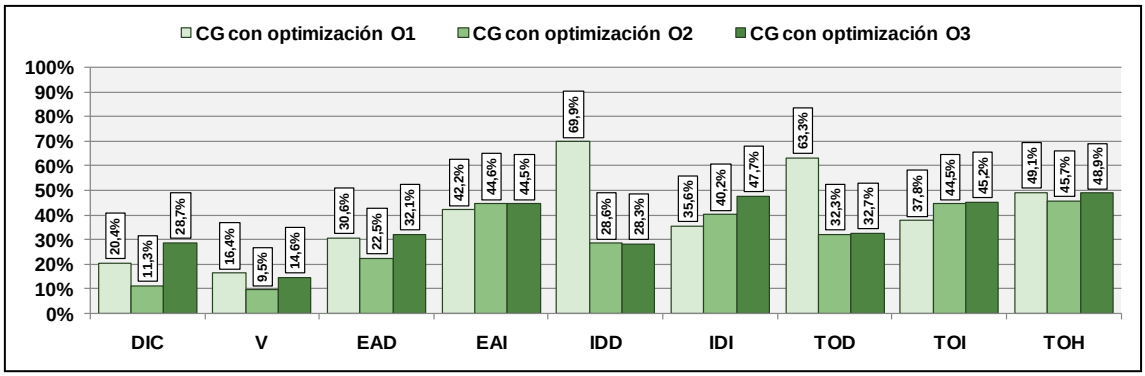

Figura 52 – Mejoría de la robustez de la aplicación CG.

En la robustez de aplicación FT, graficada en la Figura 53, notamos como van mal las reglas de transformación con acumuladores de errores de tipo punto flotante, situación que ya creíamos que pasaría por la característica de la función transformada que solamente tenía variables locales de tipo punto fijo. Logramos 64,7% de mejoría en la robustez en relación a la aplicación sin detección de fallo en nuestro mejor resultado con la regla IDI y optimización fuerte, casi el doble del mejor resultado de la regla DIC que obtuve 32,5% en su mejor resultado también con la optimización fuerte.

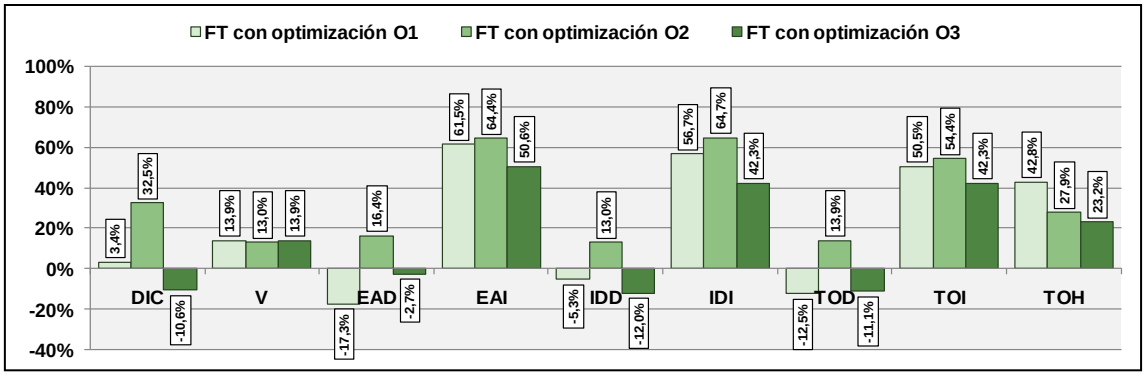

Figura 53 – Mejoría de la robustez de la aplicación FT.

Aún que no tenga sido de forma tan significativa cuanto en las dos aplicaciones presentadas anteriormente, con la aplicación SP, cuyo grafico de mejoría de la robustez está presentado en la Figura 54, seguimos mejorando la robustez de la aplicación en relación a la regla de transformación DIC. Entretanto, no hemos logrado elegir cuál de los tipos de acumuladores de errores seria más adecuado para esta aplicación.

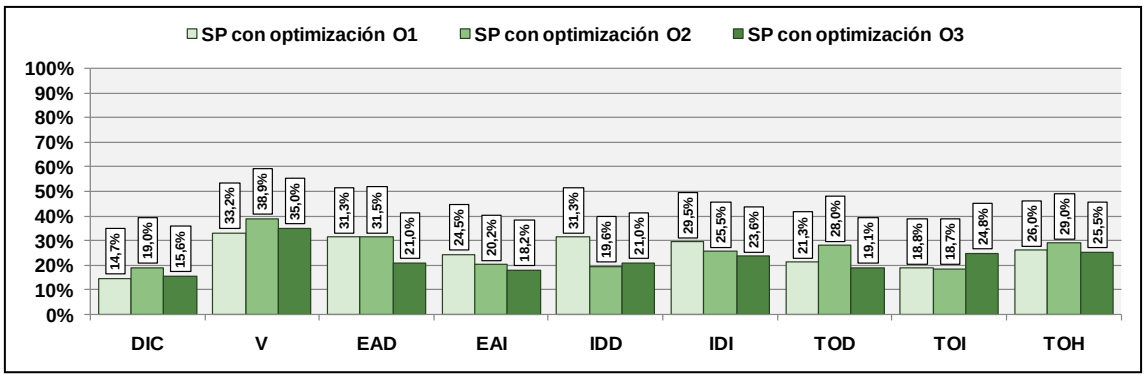

Figura 54 – Mejoría de la robustez de la aplicación SP.

Analizamos entonces los ficheros de log de nuestros experimentos de inyección de fallos de la aplicación SP y hemos notado que muchas de las inyecciones no tocaban el código transformado para mejorar la robustez.

Así, basándonos en los ficheros de log del entorno de inyección de fallos, elegimos 12 funciones a duplicar, todas usando la misma regla de transformación. Las funciones transformadas están en la Tabla 6.

Tabla 6 – Funciones transformadas en nuevo experimento con la aplicación SP.

Función	Variables punto fijo	Variables punto flotante
add	4	0
compute_rhs	4	11
lhsx	3	1
lhsy	3	1
lhsz	3	1
ninvr	3	7
pinvr	3	7
txinvr	3	14
tzetar	3	16
xsolve	7	2
ysolve	7	2
zsolve	7	2

Repitiendo, entonces, los experimentos de la aplicación SP pero de esta vez con la transformación aplicada a las 12 funciones presentadas anteriormente, hemos obtenido un resultado donde si puede inferir que las reglas con el acumulador de errores de tipo punto fijo han sobrepasado las reglas con el acumulador de errores de tipo punto flotante, situación evidente en la Figura 55.

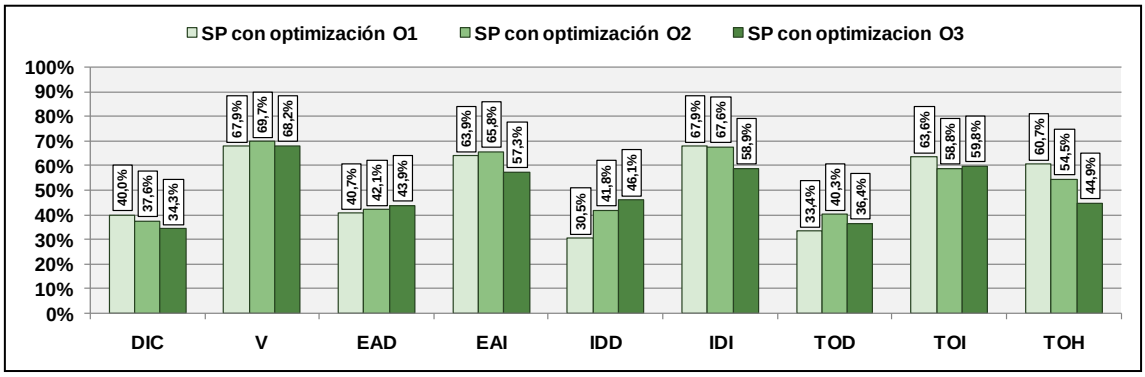

Figura 55 – Mejoría de la robustez de la aplicación SP transformando varias funciones.

6.3.5 Análisis general del MWTF

Usando los resultados obtenidos para calcular el MWTF de cada una de las aplicaciones sin normalizarlos, de modo a comparar que conjunto regla y tipo de optimización usada lograría el mejor resultado, hemos generado las graficas de cada aplicación (usando la aplicación SP con la transformación de varias funciones) en figuras que siguen.

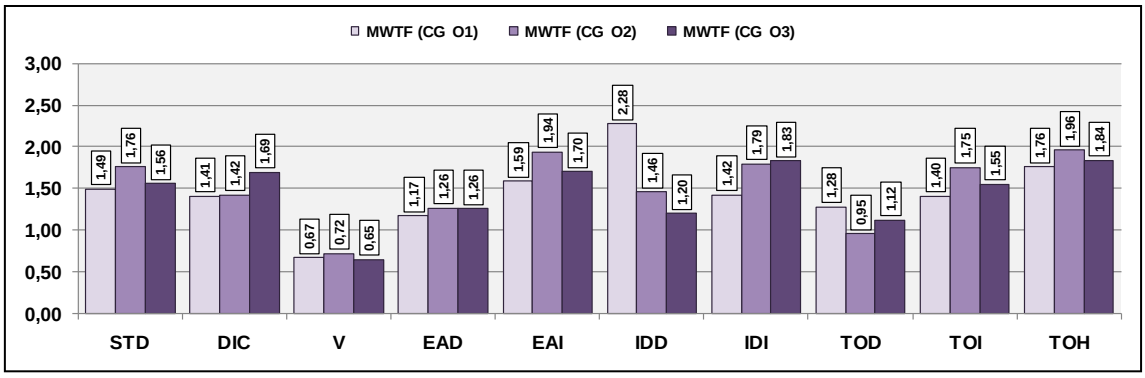

Figura 56 – MWTF de los experimentos con la aplicación CG.

En la Figura 56, notamos que el mejor MWTF general de la aplicación CG hemos obtenido con la regla de transformación IDD y con optimización moderada (O1).

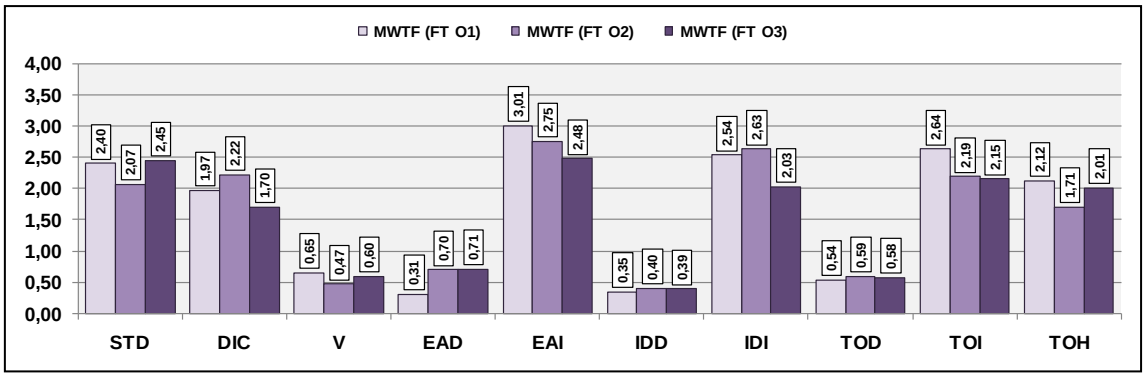

Figura 57 – MWTF de los experimentos con la aplicación FT.

En la Figura 57, notamos que el mejor MWTF general de la aplicación FT hemos obtenido con la regla de transformación EAI y también con optimización moderada (O1).

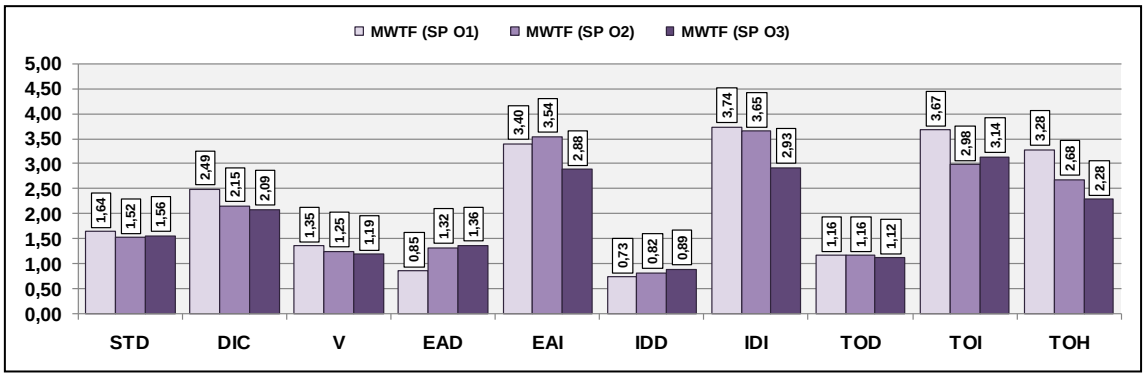

Figura 58 – MWTF de los experimentos con la aplicación SP transformando varias funciones.

Por fin, en la Figura 58, observamos que el mejor MWTF general de la aplicación SP con transformación de varias funciones hemos obtenido con la regla de transformación IDI y una vez más con optimización moderada (O1).

Seguramente, la búsqueda por mejores prestaciones con las optimizaciones más agresivas ha permitido a la optimización moderada obtener los mejores resultados absolutos de MWTF de nuestros experimentos.

6.4 Comparación con trabajos del estado del arte

Finalizando nuestro capítulo de evaluación experimental, hemos graficado los dados que hemos obtenido en el estado del arte e comparado con nuestros mejores resultados de cada una de las tres aplicaciones evaluadas en promedio. Los datos de las gráficas están en la Tabla 7.

Tabla 7 – Dados obtenidos para la comparación de trabajos.

Trabajo	Sobrecarga generada	%SDC obtenido	Mejoría en la robustez	MWTF normalizado
C2C [13]	144,0%	3,0%	76,1%	1,72
EDDI [10]	61,5%	32,4%	45,6%	1,22
SWIFT [11]	41,0%	11,7%	68,4%	2,25
EDDI Mejorado [11]	62,0%	9,2%	75,0%	2,47
FullRep (SWIFT) [12]	85,2%	4,8%	80,5%	2,77
ESoftCheck [12]	75,4%	5,6%	77,2%	2,50
SPOT [9]	319,0%	6,5%	68,1%	0,75
JGramacho[1]	77,8%	18,5%	67,7%	1,76

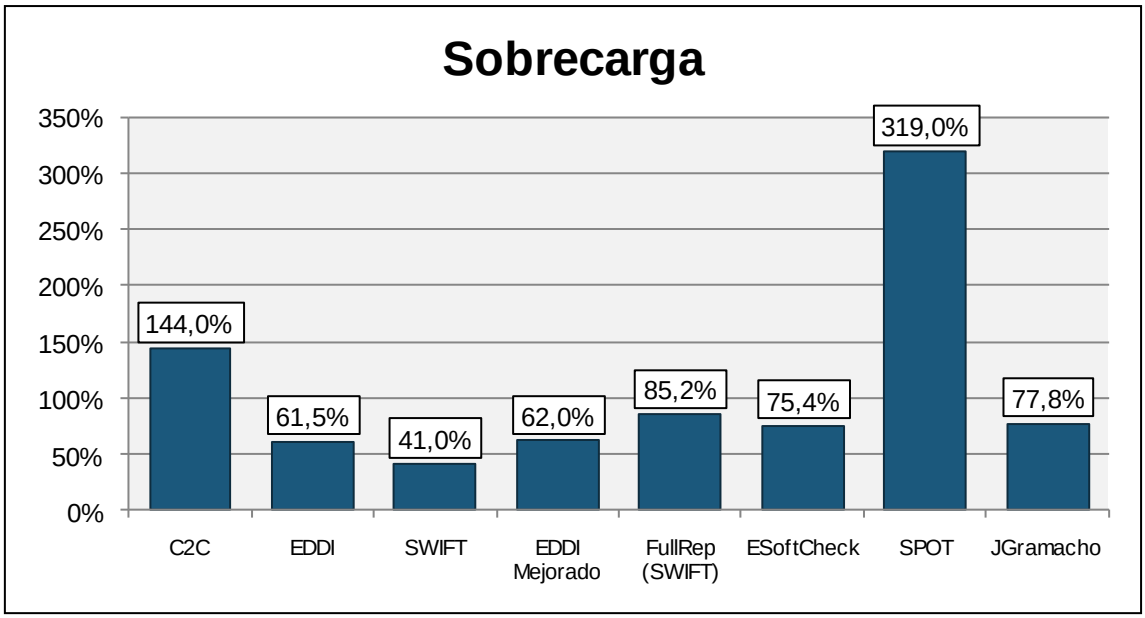

Figura 59 – Comparación general de las sobrecargas.

En relación a la sobrecarga, notamos en la Figura 59 que las propuestas que han trabajo con procesadores superescalares (EDDI, SWIFT y EDDI Mejorado) han obtenido resultados mejores que los demás y que el SPOT ha sido penalizado muchísimo por trabajar con instrumentación en dinámico. Nuestra sobrecarga en relación a las propuestas con la

[1] Resultados en promedio de este trabajo.

arquitectura x86 y compatibles ha logrado el segundo menor valor, detrás solamente del ESoftCheck.

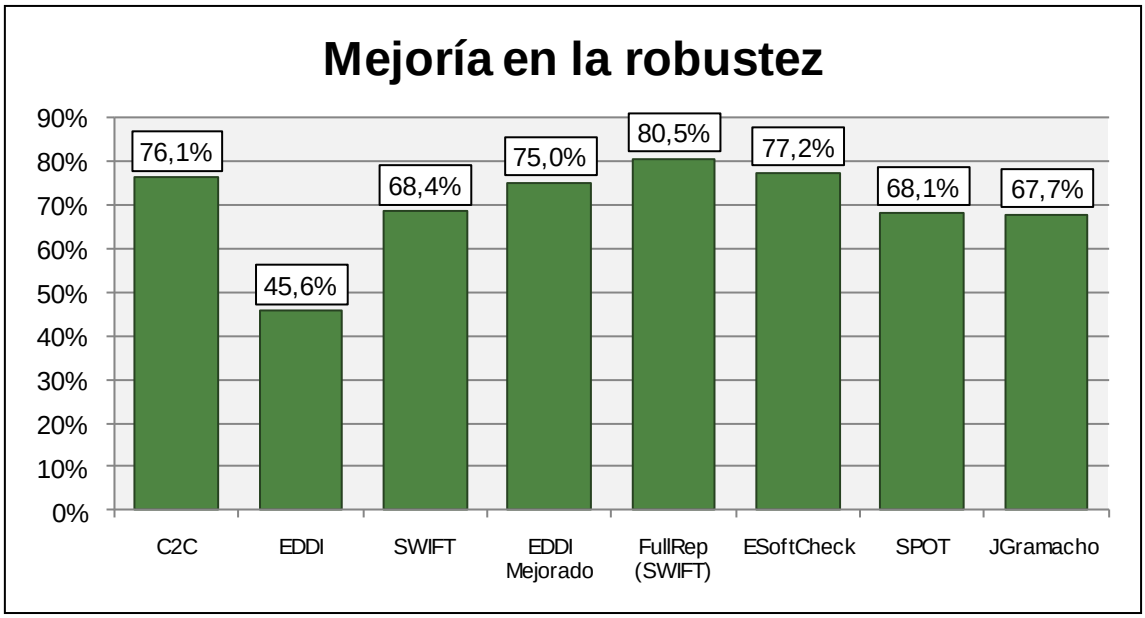

Figura 60 – Comparación general de la mejoría en la robustez.

Con respecto a la mejoría en la robustez, notamos en la Figura 60 que, a pesar de quedarnos muy próximos a los resultados obtenidos por SWIFT y SPOT, solamente no hemos logrado la menor de las mejorías porque la propuesta EDDI ha logrado el peor resultado de los estudiados.

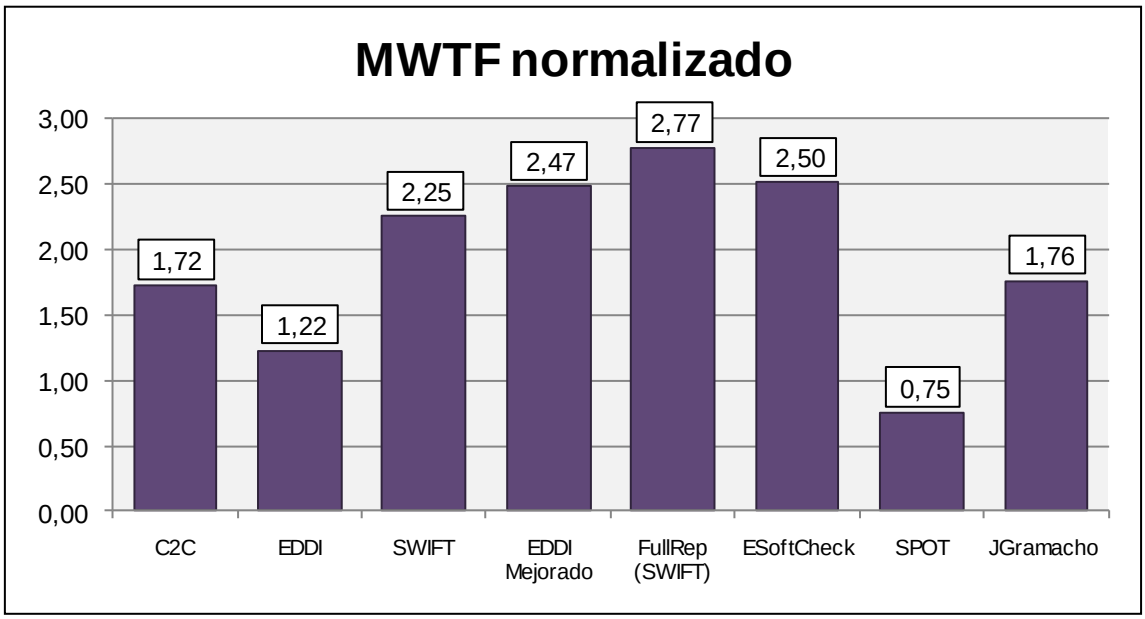

Figura 61 – Comparación general de los MWTF normalizados.

El MWTF normalizado de nuestra propuesta, según la gráfica en la Figura 61, obtuvo la quinta posición de los trabajos evaluados, quizás porque hemos probado con aplicaciones distintas de

los demás o quizás porque solamente hemos probado con tres aplicaciones y por no habernos probado una de ellas (la FT) añadiendo redundancia en más funciones.

Para mejor entender el porqué de nuestro MWTF normalizado, hemos graficado en la Figura 62 el porcentual del SDC obtenido (en promedio) por cada uno de los trabajos en sus experimentaciones con sus respectivas estrategias de añadir redundancia para la detección de errores generados por fallos transitorios.

Observamos entonces que, mismo logrando una mejora en la robustez en relación a la versión de la aplicación sin detección de fallos cercana a las otras estudiadas (en la Figura 60), nuestra aportación no ha logrado bajar la cantidad de SDC en promedio a los niveles que han logrado los otros trabajos.

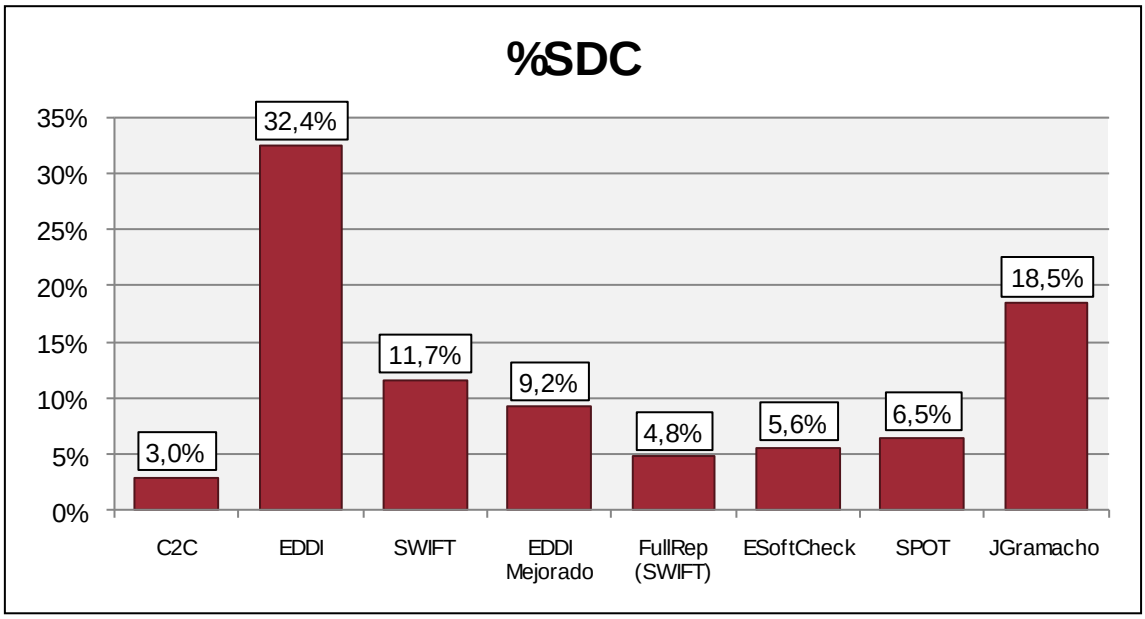

Figura 62 – Comparación general de los %SDC obtenidos.

Nuestra aportación ha logrado bajar el SDC para un nivel promedio en torno de los 18,5%, casi el doble de EDDI y más que el doble de trabajos como el EDDI mejorado y el ESoftCheck, pero creemos que para rebajar aún más el %SDC sería necesario empezar a ajustar las optimizaciones del compilador de forma a evitar que el saque parte de la redundancia añadida.

Capítulo 7 Conclusión y trabajos futuros

7.1 Conclusión

Con la evolución de los procesadores por mejor prestaciones, los chips de computadores están cada vez menos robustos frente a fallos transitorios, debido principalmente a la reducción de su tamaño, su mayor densidad de transistores y por sus componentes internos operaren a tensiones más bajas.

Esta reducción en la robustez de los procesadores está poco a poco tornando más evidente la ocurrencia de los fallos transitorios y de su peor riesgo: la corrupción silenciosa de datos, donde el fallo transitorio genera un error que no lo es notado por el sistema y puede cambiar los resultados de los cómputos de una aplicación sin que nadie lo sepa.

En la computación de altas prestaciones, este riesgo de pasar por fallos transitorios es multiplicado por la cantidad de procesadores que trabajan juntos para solucionar un único problema de forma paralela.

La literatura propone técnicas de detección de errores generados por fallos transitorios basadas en hardware, en software y también técnicas hibridas. Las alternativas basadas en software son más fáciles de aplicar en los centro de computación actuales por solamente necesitar cambios en las capas superiores (en el operativo, en el compilador o en la propia aplicación), sin necesidad de inversión en cambios de hardware.

Estas soluciones basadas en software trabajan cambiando el binario de la aplicación, usando cambios en el compilador para generar un binario más robusto, siempre intentando duplicar las operaciones basándose en el uso de los registros de la arquitectura del procesador donde las aplicaciones ejecutarán.

El problema es que la gran mayoría de los supercomputadores usan procesadores de arquitectura que no tienen tantos registros disponibles para hacer esta duplicación sin afectar las prestaciones de forma significativa, situación que todavía no pasa con las arquitecturas superescalares.

Por lo tanto, hemos elegido trabajar con cambios en el código fuente de la aplicación y dejar que el compilador haga el trabajo de planear el mejor uso de los registros del procesador. Además, buscando disminuir el impacto de las optimizaciones del compilador en el mecanismo de detección añadido y buscando también disminuir la cantidad de testes de comparación necesarios para mejorar las prestaciones, hemos propuesto el uso de un acumulador de errores.

Basándonos en las reglas básicas de transformación que hemos encontrado en la literatura, hemos propuesto un nuevo conjunto de reglas de transformación para añadir la redundancia y con ellas hemos logrado nuestro objetivo de mejorar la robustez de las aplicaciones frente a fallos transitorios sin empeorar el equilibrio entre la sobrecarga añadida por los mecanismos de detección en relación a la mejoría de la robustez.

Entretanto, comparándonos con las alternativas que hemos analizado en este trabajo, notamos que probablemente, sin tocar el compilador, será complicado llegar al nivel de los mejores trabajos que hemos evaluado.

7.2 Trabajos futuros

Como no hemos logrado llegar al nivel de los mejores trabajos que hemos evaluado, y que estos trabajos son basados en cambios a nivel de compilador, creemos que nuestra propuesta puede ser mejorada si de alguna forma se lograr entrar en el nivel del compilador y cambiarlo de forma que algunas de sus optimizaciones sean ajustadas para que no afecten negativamente la redundancia añadida a nivel de código fuente.

Otra alternativa es no trabajar con los cambios a nivel del código fuente en lenguaje C y trabar con los cambios a nivel del lenguaje intermedio usada por el compilador. Creemos que se puede haber una ganancia en relación supresión hecha por el compilador cuando hacemos el cambio en el código fuente antes de compilarlo.

Además, usar un entorno más controlable para hacer la caracterización de las aplicaciones frente a la presencia de fallos transitorios ayudaría a caracterizar con mejor precisión en cuanto mejoramos la robustez de trozos específicos de las aplicaciones (por ejemplo, de los trozos que hemos cambiado), sabiendo que un entorno más controlable debe necesitar de más tiempo para hacer las experimentaciones.

Referencias

[1] Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, p. 61, 2004.

[2] R Baumann, "Soft errors in advanced computer systems," *Design & Test of Computers, IEEE*, vol. 22, no. 3, pp. 258-266, Mayo 2005.

[3] C Constantinescu, "Dependability benchmarking using environmental test tools," *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pp. 567-571, 2005.

[4] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt, "The Soft Error Problem: An Architectural Perspective," *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 243-247, 2005.

[5] A Oliner and J Stearley, "What Supercomputers Say: A Study of Five System Logs," *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pp. 575-584, Junio 2007.

[6] Greg Bronevetsky and Bronis de Supinski, "Soft error vulnerability of iterative linear algebra methods," *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pp. 155-164, 2008.

[7] P E Dodd and L W Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 3, pp. 583-602, Junio 2003.

[8] Shubu Mukherjee, *Architecture Design for Soft Errors*,.: Morgan Kaufmann, February 2008.

[9] George A Reis, Jonathan Chang, David I August, Robin Cohn, and Shubhendu S Mukherjee, "Configurable Transient Fault Detection via Dynamic Binary Translation," *IN: PROCEEDINGS OF THE 2ND WORKSHOP ON ARCHITECTURAL RELIABILITY*, 2006.

[10] N Oh, P P Shirvani, and E J McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63-75, Marzo 2002.

[11] G A Reis, J Chang, N Vachharajani, R Rangan, and D I August, "SWIFT: software implemented fault tolerance," *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pp. 243-254, Marzo 2005.

[12] Jing Yu, Maria Jesus Garzaran, and Marc Snir, "ESoftCheck: Removal of Non-vital Checks for Fault Tolerance," *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pp. 35-46, 2009.

[13] B Nicolescu and R Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 57-62, 2003.

[14] George A Reis et al., "Design and Evaluation of Hybrid Fault-Detection Systems," *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pp. 148-159, 2005.

[15] (2010, Junio) Top500 Supercomputing Sites. [Online]. http://www.top500.org/

[16] S Mitra, Ming Zhang, N Seifert, T M Mak, and Kee Sup Kim, "Soft Error Resilient System Design through Error Correction," *Very Large Scale Integration, 2006 IFIP International Conference on*, pp. 332-337, Octubre 2006.

[17] A. Lesiak, P. Gawkowski, and J. Sosnowski, "Error Recovery Problems," *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, pp. 270-277, 2007.

[18] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 389-398, 2002.

[19] W Bartlett and L Spainhower, "Commercial fault tolerance: a tale of two systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 87-96, Enero 2004.

[20] Steven K Reinhardt and Shubhendu S Mukherjee, "Transient fault detection via simultaneous multithreading," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 25-36, 2000.

[21] S S Mukherjee, M Kontz, and S K Reinhard, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Ann Arbor*, vol. 1001, pp. 48109-2122.

[22] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P Jouppi, and James E Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pp. 470-481, 2007.

[23] James R Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190-210, 2006.