



Universitat d'Alacant
Universidad de Alicante



Co-diseño de sistemas hardware/software
tolerantes a fallos inducidos por radiación

Felipe Restrepo Calle

Tesis Doctorales

www.eltallerdigital.com

UNIVERSIDAD de ALICANTE



Universidad de Alicante

Memoria de tesis doctoral

**Co-diseño de sistemas
hardware/software tolerantes a
fallos inducidos por radiación**

Universitat d'Alacant
Universidad de Alicante
Felipe Restrepo Calle

Directores

Dr. Sergio Cuenca Asensi
Dr. Antonio Martínez Álvarez



UNIVERSIDAD DE ALICANTE

Memoria de tesis doctoral

**Co-diseño de sistemas
hardware/software tolerantes a
fallos inducidos por radiación**

Universitat d'Alacant
Felipe Restrepo Calle
Universidad de Alicante

Directores

Dr. Sergio Cuenca Asensi
Dr. Antonio Martínez Álvarez

Departamento de Tecnología Informática y Computación
TECNOLOGÍAS PARA LA SOCIEDAD DE LA INFORMACIÓN
Alicante, Septiembre 2011

A la memoria de mi padre



Universitat d'Alacant
Universidad de Alicante

*Porque tengo noble ancestro
de Don Quijote y Quimbaya,
hice una ruana antioqueña
de una capa castellana.
Por eso cuando sus pliegues
abrazo y ellos me abrazan
siento que mi ruana altaiva
me esta abrigando es el alma.*

La ruana (fragmento)
Luis Carlos González

Universitat d'Alacant
Universidad de Alicante

Agradecimientos

Son muchas a las personas que me gustaría agradecer ahora que termine esta tesis, que en realidad es una etapa de mi vida. Personas que han marcado mi vida de una, otra, o tantas maneras.

En primer lugar, quiero agradecer enormemente a Sergio y Antonio, mis directores de tesis, sin quienes este trabajo no hubiera sido posible. Por su visión, por sus capacidades y conocimientos, por su dedicación, por haber compartido conmigo su experiencia, por su infinita paciencia, por haber encontrado las palabras adecuadas en los momentos indicados para que juntos pudiéramos construir. Quiero agradecerles también, de forma muy especial, el haberme brindado su amistad. Muchas gracias a los dos de todo corazón.

Agradezco sinceramente a mis compañeros y amigos del Departamento de Tecnología Informática y Computación de la Universidad de Alicante, del grupo UniCAD y de los laboratorios de investigación, quienes han aportado enormemente a esta tesis con sus consejos, sus críticas, sus ánimos, sus bromas y todo su apoyo. Gracias LuisFe, Jorge, Rafa, Irene, Oscar, Marcelo, Rober, Pepe, Andrés, Toni, José Luis, Virgilio, Diego, José Vicente, Mora, Juan Antonio, Paco, Dani, y por supuesto a todo el equipo de personal técnico y administrativo.

Al grupo RadUS de la Universidad de Sevilla. En especial a Miguel, Hipólito y Rogelio, por el gran trabajo conjunto que hemos realizado durante los últimos años, el cual sienta las bases sólidas para continuar creando sinergias entre nuestros grupos.

Al grupo de Diseño Microelectrónico y Aplicaciones de la Universidad Carlos III de Madrid, especialmente a Luis y Almudena, por tener la iniciativa para colaborar y abrir nuevas líneas de trabajo para continuar con esta investigación en el futuro cercano.

A todo el equipo de trabajo de DUTO S.A. que desde el año 2004 ha aportado para que IRIS se convierta en un sueño hecho realidad. Pese a que no han aportado de forma directa en esta tesis, sí lo han hecho de muchas otras maneras. En especial, a Mafe y a John, cuya tenacidad, dedicación y disciplina son dignas de admiración, y con quienes espero continuar haciendo “ingeniería con función social”.

Quiero agradecer también a los amigos más cercanos de aquí, de allí y de por ahí. Los que me conocen bien, con los que he discutido acerca de sueños y proyectos, y de los que he tomado prestadas tantas enseñanzas. De aquí: a Luis

Felipe, por haber sido "pa' las que sea" en los últimos cuatro años; a Jorge, por haber estado siempre ahí cuando las cosas se complicaban (y en los momentos difíciles también) y por sus revisiones precisas a este documento; a Ir, por darle tanta alegría y "vidilla" a mi paso por Alicante (muchas gracias también por el diseño de la portada de esta tesis). A los amigos de Colombia, quiero darles las gracias por darme su confianza, por nunca haber perdido el contacto, por alegrarse con mis logros, pero sobretodo, porque cuando estamos juntos de nuevo parece que nunca nos hubiéramos alejado, muchas gracias a: el Gordo, Lore, Luza, Alejo, Mario, Renata y Sor. También a mis amiguitas que andan por ahí por el mundo, muchas gracias por entenderme y ayudarme tanto: Lina, Luisa y Mafe.

Muchas gracias también a toda mi familia. Por haber creído en mí y por apoyarme en todo momento, desde la época en que apenas podía hablar y preguntaba insistente "¿y po té?" para que me ayudaran a descubrir el mundo, hasta el día de hoy que concluyo esta tesis doctoral y afortunadamente aún continuo preguntándome el por qué de las cosas. Gracias Juan, abuelo, abuela, Jaime, Cuca, Tota, Daniel, Caro, Erin, Gus, David y Calvin.

Por último, y de manera especial, me gustaría agradecer a mi mamá, por tantas cosas que me ha dado y me ha enseñado, por creer en mí, por haberme dado la fuerza que necesité para estar lejos de todo y de todos, pero sobretodo, gracias por no haber desfallecido nunca, cuando las cosas no fueron fáciles para ninguno de nosotros.

Universitat d'Alicant
Alicante, 29 de septiembre de 2011
Felipe Restrepo Calle
Universidad de Alicante

Resumen

La progresiva miniaturización de los componentes electrónicos ha conducido a importantes avances en el desarrollo tecnológico de los microprocesadores, tal como el dramático aumento en su rendimiento. No obstante, este hecho también ha provocado algunas consecuencias adversas. Una de las más importantes es que debido a la reducción del área de los componentes electrónicos hasta escalas nanométricas, también se han reducido sus tensiones de alimentación y sus márgenes de ruido; causando que dichos componentes sean menos confiables, y por consiguiente, que los microprocesadores sean más susceptibles a *fallos transitorios* inducidos por radiación. Estos fallos intermitentes no causan daños permanentes en los circuitos, pero pueden llegar a afectar el comportamiento del sistema, al corromper la transferencia de señales o los datos almacenados. Aunque este tipo de fallos son más frecuentes en el entorno de aplicación espacial, también están presentes en una menor medida en la atmósfera e incluso a nivel terrestre.

Para superar este problema, la aplicación de redundancia hardware ha sido tradicionalmente el enfoque más utilizado. Muchas propuestas utilizan la redundancia de información para verificar la corrección de los datos almacenados memorias. Además, existen muchas técnicas basadas en la mitigación de fallos a nivel de la lógica del circuito, desde nivel de compuertas lógicas hasta el nivel de la arquitectura. Sin embargo, a pesar de que la mayoría de las propuestas basadas en hardware proveen una solución efectiva para la mitigación de fallos transitorios, por lo general estas técnicas conllevan una grave penalización al sistema en términos de recursos utilizados, consumo de potencia, área, tiempo de diseño y costes económicos; lo cual hace que su utilización no sea viable en muchos casos.

Por otro lado, durante los últimos años se han desarrollado numerosas propuestas basadas en software redundante, para brindar a los programas capacidades de detección/corrección de fallos. Estos trabajos están motivados especialmente por la necesidad de diseñar sistemas de bajo coste (utilizando componentes COTS), asegurando un nivel de confiabilidad aceptable. Muchas de las propuestas basadas en software están dirigidas a la detección de fallos. Algunas de ellas aplican a los programas la redundancia a nivel de instrucciones partiendo del código fuente de alto nivel por medio de reglas de transformación automática al código, mientras que otras aplican la redundancia de ins-

trucciones a bajo nivel (código ensamblador) con el fin de reducir el impacto en el tamaño de código, la degradación del rendimiento, y mejorar las tasa de detección de fallos. No obstante, sólo una cuantas técnicas han sido extendidas para también la recuperación del sistema. A pesar de que las técnicas basadas en software redundante son más efectivas en cuanto al coste en comparación con las técnicas hardware, no alcanzan el mismo nivel de rendimiento ni de confiabilidad de éstas ya que su aplicación implica tener que ejecutar instrucciones adicionales.

Aunque existen una amplia variedad de técnicas diferentes basadas en hardware y en software, en muchos casos la solución óptima es un punto intermedio entre estos dos extremos, una solución híbrida hardware/software. El diseño de sistemas embebidos confiables es un claro ejemplo de ello. En este caso, existen una gran cantidad de dominios de aplicación donde los factores como el consumo de potencia, el coste y el rendimiento; son tan importantes como la confiabilidad del sistema. Para esta clase de aplicaciones, se puede obtener una estrategia óptima de mitigación de fallos al combinar esquemas de protección hardware y software de diferentes técnicas. En este sentido, recientemente se han publicado algunos trabajos que muestran resultados prometedores. Sin embargo, las soluciones híbridas hardware/software existentes todavía son muy específicas y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre las restricciones de diseño y los requisitos de confiabilidad.

En este contexto, en esta tesis doctoral se propone aplicar los principios del co-diseño para diseñar a medida una estrategia híbrida de mitigación de fallos, basada en la aplicación, combinada, selectiva, e incremental de técnicas de protección hardware y software. De esta forma, será posible diseñar sistemas embebidos confiables a bajo coste, donde no sólo se satisfagan los requisitos de confiabilidad y las restricciones de diseño, sino que también se evite el uso excesivo de costosos mecanismos de protección (hardware y software).

Abstract

The ever increasing miniaturization of electronic components has led to important advances in microprocessors, such as the dramatic increase of their performance. Nevertheless, this fact has also caused adverse consequences. One of the most important is that, due to the reduction of the electronic components size to nanometric scales, voltage source levels and noise margins have also been reduced, causing electronic devices to become less reliable and, therefore, microprocessors to be more susceptible to *transient faults* induced by radiation. These intermittent faults do not provoke a permanent damage, but may affect the system behavior by altering signal transfers or stored values. Although these faults are more frequent in the space environment, they are also present to a lesser extent in the atmosphere and even at ground level.

In order to overcome this problem, applying redundant hardware has been the usual solution. Many approaches use information redundancy to verify the correctness of the data stored in memories. In addition, there are many techniques based on fault mitigation at circuit logic level, from logic gate level to architecture level. However, despite the majority of these hardware-based approaches provide an effective solution for the transient fault mitigation, in general these techniques cause a serious penalty to the system in terms of used resources, power consumption, die size, design time, and economic costs; making them unfeasible in many cases.

Moreover, in recent years, several proposals based on redundant software have been developed, providing both detection and fault correction capabilities to applications. The need for low cost solutions (using COTS components) ensuring an acceptable dependability level has especially motivated these works. Most software-based approaches are aimed to detect faults. Some of them apply redundancy to high-level source code by means of automatic transformation rules, whereas some others use instruction redundancy at low level (assembly code) in order to reduce the code overhead and performance degradation, and improve detection rates. However, only a few of these techniques have been extended to allow system recovery. Although software-based approaches are more cost effective than hardware-based ones, they cannot achieve the same performance and dependability because they have to execute additional instructions.

Despite the wide number of different fault tolerance methods based on

hardware-only and software-only, in many cases, the optimal solution is an intermediate point between these two extremes: a hybrid hardware/software solution. The design of dependable embedded systems is a clear example of this fact. In this case, there are large domains of applications where factors like power consumption, cost, and performance; are as important as dependability. Some related recent works have shown promising results. Nevertheless, current hybrid solutions are very specific, and lack the necessary flexibility to get the best trade-offs between design constraints and dependability requirements.

In this context, this doctoral thesis proposes applying the co-design principles to design a customized hybrid fault mitigation strategy, which is based on the combined, selective, and incremental application of hardware and software techniques. In this way, it will be possible to design dependable low cost embedded systems, which not only satisfy dependability requirements and design constraints, but also avoid the excessive use of costly protection mechanisms (both hardware and software).



Universitat d'Alacant
Universidad de Alicante

Índice general

Índice de figuras	XVII
Índice de cuadros	XXI
1. Introducción	1
1.1. Introducción	1
1.2. Motivación	2
1.3. Antecedentes	4
1.4. Planteamiento del Problema	6
1.5. Hipótesis	7
1.6. Objetivos	8
1.7. Metodología y plan de trabajo	8
1.8. Estructura de la tesis	9
2. Confiabilidad de los sistemas electrónicos modernos	11
2.1. Nociones básicas de confiabilidad	11
2.1.1. Averías, errores y fallos	11
2.1.2. Confiabilidad	12
2.1.3. Taxonomía de los fallos	13
2.2. Efectos de la radiación en los componentes electrónicos	16
2.2.1. Fuentes de radiación ionizante	16
2.2.2. Clasificación de los efectos de la radiación	18
2.3. Modelos de fallos	24
2.3.1. Modelos de fallos a nivel de los componentes hardware	24
2.3.2. Modelos de fallos a nivel del sistema	26
2.3.3. Modelado de fallos transitorios inducidos por radiación mediante el modelo <i>bit-flip</i>	27
2.4. Medios para alcanzar la confiabilidad	27
2.4.1. Soluciones tecnológicas	28
2.4.2. Soluciones basadas en el diseño del sistema	29
2.5. Estado actual de las técnicas de tolerancia a fallos	33
2.5.1. Técnicas de tolerancia a fallos basadas en hardware	33
2.5.2. Técnicas de tolerancia a fallos basadas en software	38

2.5.3. Técnicas híbridas de tolerancia a fallos	47
2.5.4. Técnicas de tolerancia a fallos para FPGA	49
2.6. Evaluación de la confiabilidad	51
2.6.1. Técnicas para la evaluación de la confiabilidad	51
2.6.2. Métrica para evaluar la confiabilidad	55
2.7. Conclusiones	56
3. Co-endurecimiento: co-diseño HW/SW de estrategias de endurecimiento	59
3.1. Co-diseño hardware/software	60
3.2. Co-endurecimiento hardware/software	63
3.3. Exploración del espacio de diseño de las técnicas de endurecimiento	64
3.3.1. Exploración del espacio de diseño guiada por la aplicación	69
3.4. Conclusiones	72
4. Infraestructura para co-endurecimiento	75
4.1. Herramientas requeridas para el <i>co-endurecimiento</i>	75
4.2. Entorno para el endurecimiento de software: SHE	77
4.2.1. Arquitectura genérica de microprocesadores	82
4.2.2. Núcleo de endurecimiento genérico	90
4.2.3. Endurecimiento selectivo del software	97
4.3. Herramienta para la evaluación de la confiabilidad: <i>FT-Unshades</i>	99
4.4. Conclusiones	102
5. Caso de estudio	105
5.1. Microprocesador: <i>Picoblaze</i>	105
5.1.1. Características técnicas de <i>PicoBlaze</i>	106
5.1.2. Integración de <i>PicoBlaze</i> con la infraestructura para el <i>co-endurecimiento</i>	106
5.2. Desarrollo de técnicas de endurecimiento software usando SHE	113
5.2.1. Diseño de técnicas software	114
5.2.2. Evaluación de técnicas software	121
5.3. Desarrollo de estrategias híbridas de endurecimiento	132
5.3.1. Casos de <i>co-endurecimiento</i> hardware/software	136
5.4. Conclusiones	155
6. Conclusiones y trabajos futuros	157
6.1. Conclusiones generales	157
6.2. Aportaciones	161
6.3. Publicaciones	161
6.4. Trabajo futuro	163
Bibliografía	165
Lista de acrónimos	185

Índice de figuras

2.1.	Taxonomía de los fallos según sus atributos	14
2.2.	Generación de carga en un circuito integrado como consecuencia de radiación cósmica y el pulso de corriente provocado	17
2.3.	Efectos de la radiación en los componentes electrónicos	19
2.4.	Efecto de la acción de una única partícula en un transistor MOS	21
2.5.	Clasificación de los modelos de fallos para un sistema basado en procesador	25
2.6.	Concepto de Triple Redundancia Modular – <i>TMR</i>	30
2.7.	Redundancia temporal para detectar errores transitorios	32
2.8.	Redundancia temporal para detectar errores permanentes	32
2.9.	Grafo de control de flujo de un programa	39
2.10.	Ejemplo de un programa en su versión original y trasformada mediante diversidad de datos	43
2.11.	Ejemplo de un programa con redundancia software a nivel de procedimiento	43
2.12.	Ejemplo de aplicación de reglas para detectar errores en los datos	44
2.13.	Ejemplo <i>EDDI</i>	45
2.14.	Ejemplo programa protegido con la técnica <i>SWIFT</i>	46
3.1.	Proceso tradicional de diseño de sistemas embebidos	61
3.2.	Proceso de co-diseño hardware/software de sistemas embebidos	62
3.3.	Exploración del espacio de diseño de las técnicas de endurecimiento	65
3.4.	Espacio de diseño de las técnicas híbridas hardware/software de endurecimiento	66
3.5.	Proceso de exploración del espacio de diseño de las técnicas de endurecimiento	67
3.6.	Espacio de diseño de las técnicas híbridas hardware/software de endurecimiento incluyendo enfoques selectivos	68
3.7.	Flujo de diseño del <i>co-endurecimiento</i>	70
4.1.	Infraestructura para el diseño y evaluación de técnicas híbridas de tolerancia a fallos	76

4.2. Herramientas (propias y de terceros) utilizadas para soportar la propuesta	78
4.3. Entorno para el endurecimiento software (<i>Software Hardening Environment - SHE</i>)	81
4.4. Campos que conforman una instrucción genérica	83
4.5. Tipos de instrucciones genéricas para procesadores escalares	84
4.6. Lista de operandos genéricos	84
4.7. Lista de <i>flags</i> genéricos afectados por la instrucción	85
4.8. Mapa de memoria de programa	86
4.9. Dilatación de secciones de memoria	87
4.10. Desplazamiento de secciones de memoria	88
4.11. Redistribución de secciones de memoria	88
4.12. Banco de registros genérico	89
4.13. Ejemplo de un grafo de control de flujo (CFG)	90
4.14. Ejemplo de programación con la API de endurecimiento: método <i>Harden</i>	92
4.15. Captura de pantalla del ISS	94
4.16. Esfera de replicación - SoR	98
4.17. Subdivisión de los nodos del CFG en subnodos	99
4.18. Esquema general del funcionamiento original de <i>FT-Unshades</i>	100
4.19. Esquema de funcionamiento de <i>FT-Unshades</i> usando la <i>Tabla Inteligente</i>	101
 5.1. Funcionamiento interno del <i>front-end</i> para <i>PicoBlaze</i>	109
5.2. Comparativa del informe de errores del <i>front-end</i> desarrollado versus el de <i>Xilinx KCPSM3</i>	110
5.3. Captura de pantalla del test de regresión para verificar la funcionalidad del <i>front-end</i>	113
5.4. Implementación de <i>TMR1</i> usando el API de endurecimiento	116
5.5. Programa de ejemplo endurecido con <i>TMR1</i>	116
5.6. Implementación de <i>TMR2</i> usando el API de endurecimiento	117
5.7. Programa de ejemplo endurecido con <i>TMR2</i>	118
5.8. Implementación de <i>SWIFT-R</i> usando el API de endurecimiento	119
5.9. Programa de ejemplo endurecido con <i>SWIFT-R</i>	120
5.10. Verificación funcional exitosa de un programa	124
5.11. Verificación funcional fallida de un programa	125
5.12. Verificación funcional de los programas endurecidos	126
5.13. Impacto del endurecimiento sobre el tamaño del código fuente de los programas (normalizado)	126
5.14. Impacto del endurecimiento sobre el tiempo de ejecución de los programas (normalizado)	127
5.15. Porcentajes de clasificación de fallos y MWTF normalizado para las versiones: no-endurecida (0), <i>TMR1</i> (A), <i>TMR2</i> (B) y <i>SWIFT-R</i> (C) - Campaña de prueba realizada en el ISS contra el banco de registros del microprocesador	130

5.16. Porcentajes de clasificación de fallos y MWTF normalizado para las versiones: no-endurecida (0), TMR1 (A), TMR2 (B) y SWIFT-R (C) - Campaña de prueba realizada en <i>FT-Unshades</i> contra el banco de registros del microprocesador	132
5.17. Ejemplos de SWIFT-R selectivo	134
5.18. Costes hardware para cada versión del microprocesador normalizados con respecto a <i>P0</i>	136
5.19. Campaña incremental de inyección de fallos para calibrar <i>FT-Unshades</i>	138
5.20. Impacto del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones de la aplicación <i>PID</i> (datos normalizados)	139
5.21. Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en software (microprocesador <i>P0</i>) para la aplicación <i>PID</i>	140
5.22. Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para la aplicación <i>PID</i>	142
5.23. <i>PID</i> : Ejemplo de una estrategia de endurecimiento incremental presentando el porcentaje de fallos unACE y los costes hardware normalizados	143
5.24. Impacto del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones del <i>FIR</i> (datos normalizados)	145
5.25. Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para el caso de la aplicación <i>FIR</i>	146
5.26. Costes hardware y MWTF para cada sistema endurecido selectivamente en hardware y software para el caso de la aplicación <i>FIR</i>	149
5.27. Impacto normalizado del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones endurecidas del programa <i>mmult</i>	151
5.28. Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para la aplicación <i>mmult</i>	153
5.29. Porcentajes de fallos unACE y costes de endurecimiento normalizados (coste hardware e incremento del tiempo de ejecución) para cada versión del sistema para la aplicación <i>mmult</i>	154

Índice de cuadros

4.1. Comparativo de técnicas software de detección de fallos	78
4.2. Comparativo de técnicas software de recuperación de fallos	78
5.1. Resultados obtenidos en la prueba de regresión	112
5.2. Utilización de registros en el programa <i>FIR</i>	144
5.3. Utilización de los registros usados en el programa <i>mmult</i>	150

Universitat d'Alacant
Universidad de Alicante

Capítulo 1

Introducción

Este primer capítulo introductorio resume el planteamiento del trabajo de tesis doctoral. El capítulo está organizado en ocho secciones, como se explica a continuación. La Sección 1.1 sitúa el marco de trabajo para realizar esta investigación. En la Sección 1.2 se presenta la justificación de la necesidad de llevar a cabo este trabajo. A continuación, en la Sección 1.3 se mencionan los enfoques más importantes para diseñar sistemas tolerantes a fallos inducidos por radiación. Posteriormente, en la Sección 1.4 se plantea el problema de investigación a abordar, basado en las limitaciones de las soluciones existentes. En la Sección 1.5 se expone la hipótesis de partida de la presente investigación. Por su parte, en la Sección 1.6 se presentan los objetivos de este trabajo. En la Sección 1.7 se resume la metodología utilizada para alcanzar los objetivos y se expone el plan de trabajo. Por último, en la Sección 1.8 se señala la estructura del resto de esta memoria.

1.1. Introducción

Esta memoria es el fruto del trabajo de tesis doctoral realizado en el programa de doctorado *Tecnologías de la sociedad de la información* cursado entre 2007 y 2011, en el Departamento de Tecnología Informática y Computación - DTIC, de la Universidad de Alicante (España). Trabajo realizado gracias a la beca concedida por dicha universidad en el marco de la *XII Convocatoria de Becas de Doctorado para Estudiantes Latinoamericanos*.

En esta tesis doctoral se aborda la problemática del diseño de sistemas hardware/software tolerantes a fallos inducidos por radiación. La investigación realizada está orientada a la aplicación combinada de técnicas de tolerancia a fallos basadas en hardware y software para el diseño de sistemas empotrados, que además de cumplir con los requisitos de confiabilidad, optimizan las soluciones diseñadas para el resto de los requisitos del diseño.

La presente tesis ha sido desarrollada dentro del marco de los siguientes proyectos de investigación:

- RENASER: *Efectos de la radiación sobre conductores en tecnologías nanométricas - Emulación mediante reconfiguración dinámica* (ESP2007-65914-C03-03). Proyecto financiado por el *Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica 2007* del *Ministerio de Educación y Ciencia de España*.
- RENASER+: *Análisis Integral de Circuitos y Sistemas Digitales para Aplicaciones Aeroespaciales* (TEC2010-22095-C03-01). Proyecto financiado por el *VI Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica 2008 - 2011* del *Ministerio de Ciencia e Innovación de España*.
- *Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware* (GV/2009/098). Proyecto financiado por la *Generalitat Valenciana* en *España*.

1.2. Motivación

En las últimas décadas se han realizado importantes avances en el desarrollo tecnológico de los microprocesadores. Algunos de estos avances son, entre otros: el gran aumento en rendimiento, el crecimiento desmedido del número de componentes y la cada vez mayor densidad de integración. Estos han sido posibles principalmente gracias a la progresiva miniaturización de los componentes electrónicos. No obstante, este hecho también ha provocado algunas consecuencias adversas. Debido a la reducción del área de los componentes electrónicos hasta escalas nanométricas (45nm e inferiores), también se han reducido sus tensiones de alimentación (hasta 0,9 V) y sus márgenes de ruido; causando que dichos componentes sean menos fiables. Por consiguiente, los circuitos que los utilizan son más susceptibles a varios tipos de fallos, principalmente los inducidos por radiación [Baumann, 2005a, Shivakumar et al., 2002]. Esto explica el creciente interés de la comunidad científica en la tolerancia a fallos, entendiéndose ésta como la capacidad que tiene un sistema o componente para continuar su operación normal a pesar de que se produzcan fallos en el hardware o en el software [IEEE, 2000].

Los efectos de la radiación en los componentes electrónicos pueden llegar a causar consecuencias catastróficas en sistemas de misión crítica cuya operación tiene lugar en entornos hostiles, caracterizados por la presencia de partículas de alta energía. El impacto de una de estas partículas con un componente electrónico puede producir, directa o indirectamente, ionizaciones en sus estructuras de silicio, afectando su operación de forma permanente (fallos permanentes) o temporal (fallos transitorios). Los fallos permanentes son los más graves porque implican un daño incorregible en el hardware del sistema. Mientras que los fallos transitorios acontecen físicamente cuando una de estas partículas impacta en un componente electrónico, modificando su comportamiento de forma temporal. Esto puede llegar a afectar el comportamiento del sistema completo, al corromper por ejemplo, un dato almacenado o la transferencia de una señal (*soft errors*) [Perry et al., 2007, Karnik et al., 2004].

Este tipo de fallos inducidos por radiación tradicionalmente han sido considerados como un problema concerniente a los sistemas de aplicación espacial, pues es en el espacio exterior donde abundan las partículas con muy alta energía. Sin embargo, durante las últimas décadas, este problema se ha trasladado también a los circuitos electrónicos que deben operar a nivel atmosférico e incluso a nivel terrestre [Barth et al., 2003].

A nivel atmosférico, cuando los rayos cósmicos y las partículas solares entran en la atmósfera terrestre, se atenúan debido a la interacción con los átomos de nitrógeno y de oxígeno. El proceso de atenuación produce electrones, neutrones e iones pesados, que se presentan en cantidades medibles desde 330 Km de altitud. Su densidad crece a medida que disminuye la altitud y alcanza un valor pico alrededor de los 20 Km. Por debajo de los 20 Km, la densidad de neutrones comienza a disminuir y al nivel del mar, tal densidad es de alrededor de 1/500 del valor pico [Taber, A. H. and Normand, E., 1995]. La presencia de estas partículas en la atmósfera puede llegar a afectar gravemente los sistemas computacionales que operan a este nivel, principalmente aquellos relacionados con aviación [Edwards et al., 2004].

A nivel terrestre, se pueden encontrar radiaciones producidas por el hombre (instalaciones nucleares) o radiaciones naturales (rayos cósmicos), capaces de inducir fallos transitorios en los sistemas electrónicos. A pesar de que la ocurrencia de este tipo de fallos a nivel terrestre no es tan común como a nivel aeroespacial, se han presentado numerosos incidentes alrededor de todo el mundo. Por ejemplo, en el año 2000, se informó que debido a fallos transitorios se presentaron bloqueos en los servidores (equipos de la marca *Sun*) de un número importante de sitios web, incluidos *America Online* y *eBay* [Baumann, 2002]. También, *Hewlett Packard* admitió haber tenido varios problemas en los supercomputadores de *Los Alamos Labs* debido a *fallos transitorios* [Michalak et al., 2005]. Además, *Cypress Semiconductor* ha confirmado que a finales de 2001, uno de sus clientes más importantes reportó un enorme caos en una gran compañía telefónica, y que finalmente se logró hallar que todos los problemas habían sido causados por un fallo transitorio que había provocado un bloqueo en el centro de procesamiento de datos [Ziegler and Puchner, 2004].

Como resultado de todo este escenario, se ha puesto de manifiesto la necesidad de la tolerancia a fallos inducidos por radiación en los circuitos electrónicos. Tanto así que el efecto de la radiación en las tecnologías actuales está considerado como un factor a resolver para poder continuar con el progreso tecnológico [ITRS, 2010]. Las aplicaciones del día a día de los sistemas electrónicos modernos requieren, cada vez más, mecanismos para la mitigación de estos fallos [Nicolaidis, 2011a]. Por esto, la protección de los sistemas electrónicos contra este tipo de fallos se está convirtiendo en un requisito obligatorio para un número creciente de dominios de aplicación. Entre ellos: espacial, aviación, automoción, defensa, medicina, nuclear, industrial y comunicaciones. Para cada uno de estos dominios se pueden encontrar múltiples aplicaciones nuevas donde la confiabilidad del sistema juega un papel fundamental. Tal es el caso de las siguientes aplicaciones emergentes en el campo de los sistemas empotrados en el sector aeroespacial: satélites de bajo coste [Del Corso et al., 2007],

subsistemas para el control de orientación de satélites [Arif, 2010], subsistemas de computación embarcada basados en dispositivos de lógica programable [Donohoe et al., 2007], etc.

En la actualidad los efectos de la radiación están contemplados por los estándares y normativas de diseño a nivel industrial. Diferentes comités técnicos y científicos internacionales regulan las condiciones mínimas de operación de los componentes electrónicos bajo condiciones adversas, incluyendo los efectos de la radiación. Algunas de estas normativas son:

- Aplicaciones aeroespaciales: *ESA PSS-01-609 (The Radiation Design Handbook)* [ESA, 1993].
- Aviónica: *DO-254 (Design Assurance Guideline for Airborne Electronic Hardware)* [RTCA, 2000], *IEC/TS 62396 (Process Management for Avionics Atmospheric radiation effects)* [IEC, 2006].
- Industria automotriz: *AEC-Q100 (Stress Test Qualification for Integrated Circuits)* [AEC, 2003].
- Defensa: *MIL-HSBK-817 (System Development Radiation Hardness Assurance)* [DoD, 1994].

1.3. Antecedentes

Para aquellos sistemas electrónicos donde la confiabilidad es una de las prioridades, existen numerosas técnicas disponibles para la mitigación de los fallos inducidos por radiación. Desde aquellas que implican sustituir algunos de los materiales para la fabricación de circuitos integrados [Baumann, 2005b, Baumann et al., 1995], pasando por las que proponen modificar los procesos de fabricación [Roche et al., 2004], hasta las propuestas basadas en las metodologías de diseño [Nicolaidis, 2005]. Este trabajo de investigación se enmarca dentro de éstas últimas.

Las técnicas basadas en el diseño se basan en agregar al sistema alguna funcionalidad adicional cuyo único objetivo es evitar que algún fallo que ocurra en el sistema se convierta finalmente en un error [Goloubeva et al., 2006a]. El término para referirse a estas funcionalidades adicionales se conoce como *redundancia*. Por lo general, ésta implica la adición al sistema de información de control, tiempo, recursos físicos, etc.; más allá de lo que se requeriría normalmente para la operación del sistema [Pradhan, 1996].

La redundancia hardware ha sido tradicionalmente el enfoque más utilizado para enfrentar problemas de confiabilidad en el diseño de circuitos digitales. Sin embargo, gracias al auge actual de los sistemas basados en microprocesador y a la necesidad de diseñar sistemas confiables de bajo coste, han surgido una gran cantidad de técnicas de protección fundamentadas en la utilización de software redundante. Además, más recientemente, también se han presentado propuestas basadas en enfoques de protección híbrida hardware/software.

Entre los métodos de protección basados en redundancia hardware se pueden encontrar una gran variedad de propuestas: aquellas que utilizan la redundancia de información para verificar la corrección de los datos almacenados en memoria (se pueden encontrar numerosos ejemplos en las técnicas conocidas como *Error Detection and Correction Codes* (EDACs)); y técnicas basadas en la mitigación de fallos a nivel de la lógica del circuito, desde nivel de compuertas lógicas hasta el nivel de la arquitectura.

A nivel de compuertas lógicas, se pueden proteger diferentes clases de circuitos como microprocesadores, memorias, *Application-Specific Integrated Circuits* (ASICs) y circuitos reprogramables, entre otros. Se trata de reemplazar las celdas de almacenamiento tradicionales (celdas *Static Random Access Memory* (SRAM), *flip-flops*, *latches*) por celdas de almacenamiento protegidas (*endurecidas*). En [Gusmão-de Lima, 2000] se puede encontrar un estudio comparativo de diferentes tipos de celdas de almacenamiento endurecidas.

Con respecto a la arquitectura, se aplican técnicas que van desde la replicación de estructuras básicas hasta la utilización de componentes más complejos. Las primeras suelen duplicar o triplicar estructuras de bajo nivel y adicionar votadores por mayoría para detectar/corregir el fallo (*Dual Modular Redundancy* (DMR) y *Triple Modular Redundancy* (TMR) [Von-Neumann, 1956]). Las segundas añaden nuevas unidades funcionales [Austin, 1999], co-procesadores [Mahmood and McCluskey, 1988], replican el sistema completo (por ejemplo, usando dos FPGA [Kubalik and Kubatova, 2008]), o incluso proponen el aprovechamiento de la multiplicidad de bloques hardware adicionales, disponibles en arquitecturas multi-hilo/multi-núcleo, para implementar la redundancia [Vijaykumar et al., 2002, Mukherjee et al., 2002, Gomaa et al., 2003].

Otras propuestas más recientes de redundancia hardware publicadas en la literatura, proponen el endurecimiento selectivo de algunos módulos del sistema. Por ejemplo: aquellas cuyo objetivo es proteger únicamente las partes más vulnerables del circuito [Samudrala et al., 2004]; otras que intentan reducir la degradación del rendimiento ocasionada por la protección, mediante la aplicación de forma parcial de técnicas de redundancia multi-hilo/multi-núcleo [Parashar et al., 2006, Reddy et al., 2006]; o las que utilizan estructuras hardware simples para proteger los valores almacenados (aquellos que almacenan *narrow values*), replicando únicamente una parte de sus bits (los utilizados), en su parte disponible [Ergin et al., 2009].

A pesar de que la mayoría de las propuestas basadas en hardware proveen una solución efectiva para la mitigación de fallos transitorios, por lo general estas técnicas conllevan una grave penalización al sistema en términos de recursos utilizados, consumo de potencia, área, tiempo de diseño, tiempo de procesamiento y costes económicos. Por esta razón, su aplicación no es viable en muchos casos.

Por otro lado, durante los últimos años se han desarrollado numerosas propuestas basadas en software redundante para dotar a los programas capacidades de detección/corrección de fallos. Estos trabajos están motivados por la necesidad de diseñar sistemas de bajo coste (utilizando componentes *Commercial Off The Shelf* (COTS)) asegurando un nivel de confiabilidad aceptable.

ble [Nicolescu et al., 2004, Oh et al., 2002a, Rebaudengo et al., 2003].

Aunque la redundancia software puede aplicarse con cuatro niveles de granularidad diferentes: instrucción, bloque de instrucciones, procedimiento y programa; las técnicas que están basadas en redundancia a nivel de instrucción obtienen mejores resultados en cuanto a detección e incluso recuperación de fallos [Goloubeva et al., 2006a].

Muchas de las propuestas basadas en software están dirigidas solamente a la detección de fallos. Algunas aplican la redundancia a los programas a nivel de instrucciones por medio de transformaciones automáticas del código fuente en un lenguaje de alto nivel (normalmente lenguaje C) [Rebaudengo et al., 2001]. Otras propuestas aplican la redundancia de instrucciones a nivel de código ensamblador con el fin de reducir el impacto en el tamaño de código y el tiempo de ejecución del programa tras el endurecimiento, y también, para mejorar la tasa de detección de fallos [Oh et al., 2002c, Oh et al., 2002b, Reis et al., 2005b]. Algunas de estas técnicas han sido extendidas para permitir no sólo detectar los fallos, sino también la recuperación del sistema [Rebaudengo et al., 2004, Reis et al., 2007].

A pesar de que las técnicas basadas en software redundante son menos costosas en comparación con las técnicas hardware, no alcanzan el mismo nivel de rendimiento ni de confiabilidad de éstas. Esto se debe principalmente a que su aplicación implica tener que ejecutar instrucciones adicionales en los programas [Azambuja et al., 2010a]. No obstante, es interesante mencionar que algunas de estas técnicas basadas en software ya han sido utilizadas en satélites y sistemas que participan en misiones espaciales [Pignol, 2010].

Dentro de los sistemas embebidos, existen una gran cantidad de dominios de aplicación donde factores como el consumo de potencia, el coste y el rendimiento son tan importantes como la confiabilidad del sistema. Para esta clase de aplicaciones, la solución óptima suele encontrarse en un punto intermedio entre las técnicas puramente hardware o software. Es decir, una solución híbrida de mitigación de fallos que combina esquemas de protección hardware y software. En este sentido, recientemente se han publicado algunos trabajos que muestran resultados prometedores [Bernardi et al., 2010, Reis et al., 2005a]. Sin embargo, las soluciones propuestas todavía son muy específicas y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre los requisitos generales del diseño y los específicos de confiabilidad.

De los trabajos mencionados en este apartado, aquellos que tienen mayor relevancia para esta investigación, serán descritos con un mayor nivel de detalle en el siguiente capítulo.

1.4. Planteamiento del Problema

De acuerdo con lo expuesto anteriormente, tanto en la sección de motivación como en el apartado de antecedentes, es interesante resaltar lo siguiente:

-
- La continua miniaturización de los componentes electrónicos ha ocasio-

nado que los sistemas digitales actuales sean más susceptibles a fallos transitorios inducidos por radiación. Este hecho ha causado que la importancia de diseñar sistemas digitales tolerantes a fallos haya ido en aumento en las últimas décadas.

- Pese a que los fallos inducidos por radiación eran considerados usualmente como un problema propio de los sistemas de misión crítica que operan a nivel espacial, desde hace varios años, este problema se ha trasladado también a los sistemas confiables desplegados a nivel atmosférico e incluso a nivel terrestre. Esto hace aún más evidente la necesidad de mitigar los efectos de la radiación en los circuitos electrónicos.
- Las técnicas de diseño para la mitigación de fallos están basadas en la incorporación de módulos redundantes en el sistema.
- Los enfoques basados en redundancia hardware ofrecen una solución efectiva para la mitigación de fallos. Sin embargo, provocan un mayor consumo de recursos, potencia, mayores costes económicos e incremento en el tiempo de desarrollo. Por lo cual, estos enfoques no resultan adecuados en todos los casos.
- Gracias al auge de los sistemas digitales basados en microprocesador y a la búsqueda de técnicas de mitigación menos costosas, han surgido numerosas propuestas basadas en redundancia software. La principal ventaja de estos enfoques es que están libres de costes hardware y permiten la utilización de componentes COTS. No obstante, su utilización tiene un mayor impacto espacial y temporal en los programas.
- Existen evidencias de que mediante la utilización de técnicas de protección híbridas hardware/software es posible encontrar un punto medio entre las soluciones hardware y software donde se aprovechen las bondades que ofrecen ambos enfoques por separado. Sin embargo, las técnicas híbridas existentes actualmente son muy específicas y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre confiabilidad, rendimiento, coste, y otros parámetros asociados a los requisitos generales del diseño.

1.5. Hipótesis

Las técnicas actuales de mitigación de fallos transitorios inducidos por radiación están dirigidas a un amplio rango de aplicaciones lo cual puede occasionar algunos inconvenientes al implementar sistemas concretos. Esto es, la aplicación de alguna de estas técnicas puede suponer el incremento en el coste desmedido, o bien, perjudicar el rendimiento del sistema de forma inaceptable.

La hipótesis de partida de esta investigación se fundamenta en que la aplicación combinada y selectiva de técnicas de protección hardware y software

permitirá diseñar sistemas que, además de cumplir con las exigencias de confiabilidad, optimizan el resto de los requisitos de diseño.

1.6. Objetivos

El objetivo general de la investigación es la propuesta y validación de una metodología de desarrollo de estrategias híbridas de mitigación de fallos, mediante la combinación selectiva, incremental y flexible de enfoques hardware y software.

Para conseguir este objetivo se han planteado una serie de objetivos específicos:

- Realizar un estudio detallado del estado actual y las soluciones existentes en cuanto al diseño de sistemas electrónicos tolerantes a fallos. Asimismo, estudiar las diferentes métricas para evaluar los parámetros de confiabilidad de dichos sistemas.
- Desarrollar una metodología, basada en los principios del co-diseño hardware/software, para el diseño de técnicas híbridas de mitigación de fallos.
- Desarrollar un conjunto de herramientas que soporten la propuesta. Estas herramientas deben permitir la implementación, aplicación y evaluación de las técnicas diseñadas de manera flexible.
- Validar la aplicabilidad y la flexibilidad de la propuesta.

1.7. Metodología y plan de trabajo

La metodología empleada para llevar a cabo la presente investigación se basa en el método de investigación científica, más concretamente en el método hipotético deductivo. Por lo cual, a partir del estudio profundo del estado actual del diseño de sistemas tolerantes a fallos, se ha identificado un problema que requiere investigación. Posteriormente, se ha formulado una hipótesis de partida para dar solución a dicho problema, la cual se desarrollará y se validará por medio de la experimentación a lo largo de esta memoria.

A continuación se expone el plan de trabajo para llevar a cabo esta investigación, el cual se desprende de los objetivos específicos planteados anteriormente:

1. Estudio del estado del arte y antecedentes para demostrar la existencia del problema en cuestión y la idoneidad de la investigación. Esto incluye:
 - Nociones básicas de tolerancia a fallos.
 - Fallos transitorios ocasionados por radiación.
 - Técnicas de tolerancia a fallos.

- Evaluación de la confiabilidad.
2. Propuesta para el co-diseño a medida de sistemas hardware/software tolerantes a fallos, donde las decisiones de diseño sean guiadas principalmente por los requisitos de diseño y confiabilidad de cada aplicación.
 3. Diseño e implementación de una plataforma de endurecimiento que soporte el enfoque propuesto.
 - Desarrollar las herramientas software necesarias para el desarrollo de técnicas software de endurecimiento y su correspondiente implementación de forma automática en los programas.
 - Desarrollar una herramienta de inyección de fallos basada en simulación mediante software para evaluar de forma preliminar las técnicas de endurecimiento implementadas.
 - Incorporar a la plataforma de endurecimiento una herramienta hardware de emulación de fallos que permita evaluar el nivel de confiabilidad del sistema hardware/software de forma más realista.
 4. Aplicar la metodología propuesta a diferentes casos de estudio para validar su aplicabilidad y flexibilidad. Se contemplarán aspectos como:
 - Diferentes requisitos de diseño y confiabilidad.
 - Endurecimiento software.
 - Endurecimiento hardware.
 - Endurecimiento hardware/software.
 - Endurecimiento selectivo.
 - Análisis de compromisos de los resultados obtenidos.
 5. Determinar las principales novedades aportadas y las líneas de trabajo futuro a seguir.

1.8. Estructura de la tesis

La memoria de esta tesis doctoral está dividida en seis capítulos. A continuación se presenta cada uno de ellos:

- Capítulo 1. *Introducción*: este primer capítulo introductorio resume el planteamiento del trabajo de tesis doctoral.
- Capítulo 2. *Confiabilidad de los sistemas electrónicos modernos*: en este capítulo se define el marco teórico y conceptual que enmarca esta investigación, y además, se presenta una revisión del estado actual de las técnicas de tolerancia a fallos.

- Capítulo 3. *Co-endurecimiento: co-diseño hardware/software de estrategias de endurecimiento*: presenta el enfoque propuesto, el cual se fundamenta en los principios del co-diseño hardware/software, y consiste en el diseño a medida de estrategias híbridas de mitigación de los efectos de la radiación en los sistemas electrónicos.
- Capítulo 4. *Infraestructura para el co-endurecimiento*: se presentan los detalles de diseño, implementación y funcionamiento de las herramientas que conforman la infraestructura que permite llevar a cabo las tareas propias del co-endurecimiento.
- Capítulo 5. *Caso de estudio*: en este capítulo se presenta un completo caso de estudio para validar la propuesta por medio de resultados experimentales.
- Capítulo 6. *Conclusiones y trabajos futuros*: finalmente, este capítulo recoge las conclusiones extraídas del trabajo realizado, presenta sus principales aportaciones, y también, las futuras líneas de investigación que se desprenden de la realización del mismo.



Universitat d'Alacant
Universidad de Alicante

Capítulo 2

Confiabilidad de los sistemas electrónicos modernos

Con el objetivo de establecer las bases que permiten llevar a cabo este trabajo de tesis, en este capítulo se define el marco teórico y conceptual que enmarca esta investigación. Está dividido en siete secciones. La Sección 2.1 presenta un marco teórico y conceptual relacionado con los conceptos básicos de confiabilidad y la taxonomía de los fallos. Posteriormente, en la Sección 2.2 se hace énfasis en los fallos causados por los efectos de la radiación en los sistemas electrónicos. La Sección 2.3 presenta los principales modelos utilizados para la representación de los fallos. Posteriormente, en la Sección 2.4 se describen los medios utilizados para alcanzar la confiabilidad en un sistema electrónico. Por su parte, en la Sección 2.5 se reúnen las principales estrategias de tolerancia a fallos disponibles en la actualidad. Seguidamente, en la Sección 2.6 se presentan las métricas y las técnicas disponibles en la literatura para evaluar la confiabilidad. Por último, la Sección 2.7 concluye el capítulo con las consideraciones más significativas encontradas en el estudio realizado del estado actual de la técnica.

2.1. Nociones básicas de confiabilidad

En esta sección se presentan algunos conceptos básicos sobre la temática de la confiabilidad de los sistemas electrónicos. Resulta necesario definir algunos términos que serán de gran utilidad para establecer las bases conceptuales necesarias y abordar las discusiones en el resto de este documento. Las definiciones presentadas provienen principalmente de [Avizienis et al., 2004], [Laprie et al., 1992], y [Geffroy and Motet, 2002].

2.1.1. Averías, errores y fallos

- **Avería.**

Una avería (*failure* en inglés) se puede definir como la interrupción del desempeño o inhabilidad del sistema para llevar a cabo la función para la que ha sido diseñado, durante un periodo de tiempo específico y bajo ciertas condiciones ambientales.

■ **Fallo.**

Un fallo (*fault*) es la causa de una avería. Un fallo puede ser de dos tipos:

- **Durmiente o pasivo:** cuando está presente en el sistema pero no afecta a su funcionamiento.
- **Activo:** cuando afecta al funcionamiento del sistema.

■ **Error.**

Un error es un fallo activo, es decir, es la causa de una avería que compromete al sistema. Un error causa una avería tan pronto como se propaga desde la parte interna del sistema a sus salidas. Una vez que un fallo ha sido activado como error en un módulo del sistema, se puede propagar a través de múltiples caminos hasta alcanzar las salidas del sistema.

Sin embargo, es importante mencionar que no todos los fallos se convierten en errores y no todos los errores se convierten en averías. La propagación de fallos depende de múltiples factores tales como: el módulo del sistema donde se originó el fallo, la estructura del sistema y la secuencia de entradas del sistema.

Un fallo puede permanecer pasivo en el sistema hasta que se convierta en un error. En este caso hay dos conceptos importantes a tener en cuenta:

- **Activación inicial:** es la primera ocurrencia del error provocado por un fallo.
- **Latencia:** tiempo medio entre la ocurrencia del fallo y su activación inicial como error.

2.1.2. Confiabilidad

La confiabilidad, o en inglés *dependability*, es el objetivo que se persigue durante el diseño de un sistema tolerante a fallos. En [Avizienis et al., 2004], se encuentran dos definiciones. La primera define el término *confiabilidad*, como la habilidad de un sistema para entregar un servicio en el que se puede confiar justificablemente. La segunda definición, más objetiva, establece que la *confiabilidad* de un sistema es su habilidad para evitar que las averías sean más frecuentes y más severas de lo aceptable. Este concepto engloba los siguientes atributos del sistema:

- **Disponibilidad (availability):** determina cuan listo está un sistema para entrar en funcionamiento de forma correcta. Puede ser definido como la probabilidad de que un sistema sea capaz de entregar servicio correctamente en un intervalo de tiempo dado.
-

- **Fiabilidad (reliability):** indica la capacidad de proveer continuidad durante la prestación del servicio de forma correcta. Puede ser definida también como la probabilidad de que un sistema funcione correctamente durante un intervalo de tiempo dado, bajo cierto conjunto de condiciones de trabajo.
- **Seguridad (safety):** ausencia de consecuencias catastróficas para los usuarios y el entorno cuando se produce un error. La seguridad se mide como la probabilidad de que un sistema funcione correctamente y en caso de que falle lo haga de un modo seguro.
- **Integridad (integrity):** es la capacidad de evitar alteraciones no apropiadas en el sistema. Como se sugiere en [Storey, 1996], podemos definir dos tipos de integridad:
 1. **Integridad del sistema:** definida como la habilidad del sistema para detectar fallos durante su operación e informar a un operador humano.
 2. **Integridad de los datos:** definida como la capacidad de un sistema de prevenir daños en su base de datos y de detectar, y posiblemente corregir, errores que ocurran como consecuencia de los fallos.
- **Mantenibilidad (maintainability):** es la capacidad de un sistema de experimentar modificaciones y reparaciones. Dicho de otro modo, podemos definir el término *mantenimiento* como la acción ejecutada para mantener al sistema internamente, o hacerlo retornar a las condiciones de operación para las cuales ha sido diseñado. La mantenibilidad es la capacidad del sistema para ser sometido a mantenimiento.

2.1.3. Taxonomía de los fallos

A pesar de que los atributos a considerar para caracterizar un fallo varían de unos autores a otros en la literatura, todos los fallos que pueden afectar un sistema durante su vida se pueden clasificar de acuerdo a ocho características elementales de acuerdo a [Johnson, 1989] y [Avizienis et al., 2004]. La Figura 2.1 presenta esta clasificación.

A continuación se describen detalladamente cada uno de estos atributos:

- **Fase de ocurrencia.** Un fallo puede producirse en cualquiera de las etapas del ciclo de vida del sistema. Así se distinguen entre fallos ocasionados en la fase de diseño, que incluye desde la etapa de especificación (errores de diseño) hasta fallos en los componentes durante el proceso de fabricación; o fallos ocasionados durante la fase de operación normal del sistema.
- **Fronteras del sistema.** Un fallo se puede originar internamente dentro del sistema o externamente a él. Los fallos con origen interno pueden estar ocasionados por el deterioro físico de los componentes del sistema.

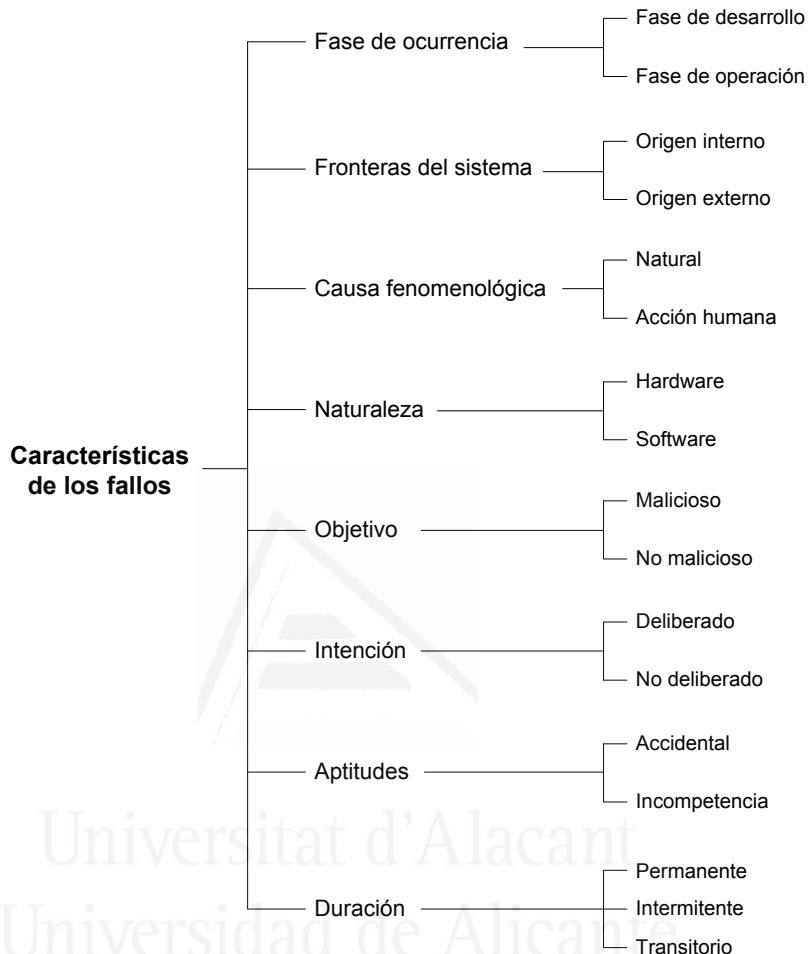


Figura 2.1: Taxonomía de los fallos según sus atributos

Los que tienen origen externo pueden propagarse al sistema por medio de interacciones o interferencias.

- **Causa fenomenológica.** Este atributo especifica si el fallo se ha originado a causa de fenómenos naturales sin la participación humana; o si por el contrario, es el resultado de acciones por parte del ser humano.
- **Naturaleza.** Con respecto a su naturaleza, los fallos se pueden clasificar en función de la parte del sistema a la que afectan. Así, se pueden distinguir fallos hardware o software. Los primeros se pueden originar en algún componente físico o simplemente afectar el hardware del sistema. Éstos a su vez se pueden clasificar en fallos digitales o analógicos. Los fallos software pueden afectar a los programas o a los datos.

- **Objetivo.** Se pueden distinguir de acuerdo al propósito que tenga la persona que interactúa con el sistema. Los fallos maliciosos se introducen al sistema con el claro propósito de causar algún daño a éste. Aquellos fallos originados sin dicha intención, se conocen como fallos no maliciosos.
- **Intención.** Los fallos pueden ser deliberados o no deliberados. Los fallos deliberados se deben a malas decisiones, que son acciones intencionadas y equivocadas. Los no deliberados se deben principalmente a errores humanos sin intención por parte del desarrollador o el operador del sistema.
- **Aptitudes.** La mayoría de fallos deliberados o no deliberados suelen ser accidentales, siempre y cuando no estén provocados con mala intención. Sin embargo, no todos estos fallos pueden ser considerados del mismo modo. Algunos están provocados por personas que carecen de las competencias profesionales para llevar a cabo el trabajo de forma correcta.
- **Duración.** Existen tres tipos de fallos dependiendo de su duración. El fallo es permanente si está localizado en el espacio y tiene duración ilimitada. Un fallo se clasifica como intermitente si además de tener una duración limitada se repite en el tiempo en la misma posición de forma no periódica. El fallo es transitorio si aparece durante un corto periodo de tiempo y luego desaparece.

De esta forma, un fallo queda determinado por la combinación de características que presenta. No obstante, no todas las combinaciones son posibles. Por ejemplo un fallo originado por causas naturales no puede presentar una intención deliberada y maliciosa.

Los fallos permanentes se deben principalmente a defectos de fabricación y al envejecimiento de los componentes físicos del sistema, aspectos que llevan asociada una localización espacial. Las causas y mecanismos que originan fallos intermitentes son similares a los de los fallos permanentes. De hecho algunos mecanismos por envejecimiento, en ocasiones, pueden producir fallos intermitentes hasta que se produce un fallo permanente, de forma que un elevado porcentaje de los fallos intermitentes aparecen como preludio de fallos permanentes.

Por otra parte, los fallos transitorios (en inglés *transient faults*) se deben principalmente a causas ambientales que no producen un defecto físico. Este tipo de fallos puede estar causado, por ejemplo, por interferencias electromagnéticas o por radiación cósmica.

Debido a la complejidad del problema, el estudio de soluciones para cada una de las etapas del ciclo de diseño (especificación, diseño, fabricación y operación) se ha desarrollado como una temática específica. La tolerancia a fallos se centra en estudiar el problema de los fallos originados durante la fase de operación. Como consecuencia del desarrollo de la tecnología, los circuitos digitales son cada vez más sensibles a las condiciones externas, aumentando el número de fallos transitorios que afectan a los sistemas electrónicos modernos durante su fase de operación. Esta investigación se centra concretamente

en el estudio de fallos transitorios, inducidos por radiación, durante la fase de operación del sistema, y que afectan a los recursos hardware que lo componen.

En la siguiente sección, se presenta la problemática actual de los efectos de la radiación en los componentes electrónicos modernos, y los diferentes tipos de fallos asociados a ella.

2.2. Efectos de la radiación en los componentes electrónicos

Como se expuso en la Sección 1.2, con la introducción de las actuales tecnologías y la continua miniaturización de los componentes electrónicos, también se han reducido sus tensiones de alimentación y sus márgenes de ruido considerablemente. Este hecho ha ocasionado que los actuales sistemas electrónicos modernos resulten ser más propensos a fallos ocasionados por fenómenos naturales externos [Baumann, 2005a, Shivakumar et al., 2002]. El progreso hacia las dimensiones nanométricas agrava el problema con cada nueva tecnología de fabricación, estando reconocido en la actualidad como un factor que puede impedir el progreso tecnológico si no se toman ciertas precauciones [ITRS, 2010].

Existen varios factores ambientales que pueden llegar a ocasionar fallos transitorios y alteraciones en el comportamiento de componentes electrónicos. Algunos ejemplos de estos son: variaciones en la temperatura, variaciones en la tensión de alimentación, interferencias electromagnéticas, y los efectos inducidos por radiación. En esta investigación se abordan estos últimos.

Los fallos inducidos por radiación tradicionalmente han sido considerados como un problema exclusivo de los sistemas de aplicación espacial, pues es en el espacio exterior donde abundan las partículas con muy alta energía. Sin embargo, durante las últimas décadas, este problema se ha trasladado también a los circuitos electrónicos que deben operar a nivel atmosférico e incluso a nivel terrestre (a nivel del mar) [Barth et al., 2003].

2.2.1. Fuentes de radiación ionizante

En [Baumann, 2001] y en [Wang and Agrawal, 2008] se resumen las principales características y las fuentes de los tres tipos principales de radiación que pueden llegar a inducir fallos en un dispositivo semiconductor. Estos son:

- Partículas alfa. Estas son uno de las diferentes radiaciones que se pueden emitir cuando el núcleo de un isótopo inestable decae a un estado de energía menor [May and Woods, 1979, Saihalasz et al., 1982].
- Neutrones con alta energía procedentes de rayos cósmicos [Normand, 1996, Gossett et al., 1993, Ziegler and Lanford, 1981].
- La interacción entre los neutrones en equilibrio térmico provenientes de rayos cósmicos y el isótopo de Boro ^{10}B , este último presente en dispositi-

vos que contienen *Borophosphosilicate glass* (BPSG) [Baumann et al., 1995, Baumann and Smith, 2001].

La radiación cósmica está constituida por una serie de partículas cargadas (protones, partículas alfa, iones pesados, como por ejemplo núcleos de hierro) y ondas altamente energéticas que provienen del espacio exterior. Esta es una radiación ionizante, lo que significa que interacciona con la materia a la que llega con la suficiente energía como para provocar la pérdida de electrones en los átomos, ionizando las moléculas. Otro efecto que puede resultar de dicha interacción es el desplazamiento de los átomos fuera de sus posiciones en la estructura cristalina, modificando las propiedades del material. Sin embargo, para las energías incidentes típicas la ionización es el mecanismo de absorción dominante, por lo que es la causa principal de los fallos originados en los sistemas digitales [Schrimpf, 2007].

A causa del carácter ionizante de la radiación cósmica, cuando dicha radiación incide sobre un semiconductor, genera pares electrón-hueco. El campo eléctrico en los circuitos recoge a los portadores creando un pulso de corriente de magnitud similar a la de operación de los transistores, lo que puede provocar un mal funcionamiento del componente y producir un fallo en el circuito. La Figura 2.2 ilustra este proceso [Baumann, 2005b].

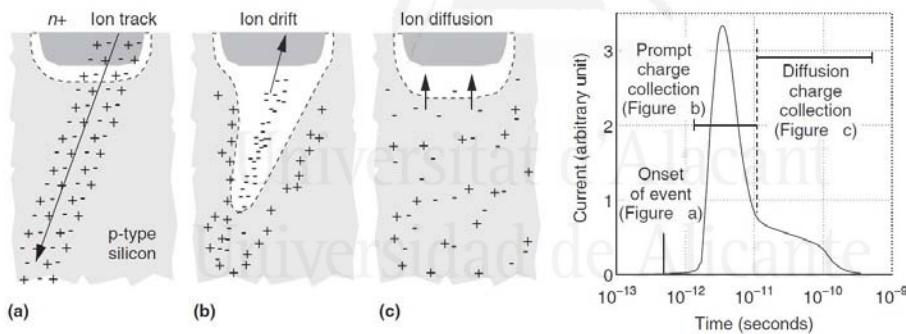


Figura 2.2: Generación de carga en un circuito integrado como consecuencia de radiación cósmica y el pulso de corriente provocado

Cuando los rayos cósmicos y las partículas solares entran a la atmósfera terrestre, se atenuan debido a la interacción con los átomos de nitrógeno y de oxígeno, generando cascadas de partículas secundarias. El proceso de atenuación produce electrones, neutrones e iones pesados, que se presentan en cantidades medibles desde 330 Km de altitud. Su densidad crece a medida que disminuye la altitud y alcanza un valor pico alrededor de los 20 Km. Por debajo de esta altura, la densidad de neutrones comienza a disminuir y al nivel del mar, tal densidad es de alrededor de 1/500 del valor pico [Taber, A. H. and Normand, E., 1995]. La presencia de estas partículas en la atmósfera puede llegar a afectar gravemente a los sistemas computacionales

que operan a este nivel, principalmente aquellos relacionados con aviación [Edwards et al., 2004].

Las partículas secundarias con suficiente energía para continuar atravesando la atmósfera interaccionan a su vez generando nuevas partículas, y así sucesivamente hasta que algunas partículas (menos del 1 % del flujo de partículas inicial) alcanzan la superficie terrestre. Dentro del flujo de radiación cósmica que alcanza la superficie terrestre, los neutrones son uno de los componentes más numerosos. Los neutrones son partículas sin carga por lo que no generan directamente la ionización de un material, como el silicio en los circuitos integrados. No obstante, debido a su masa, generan iones de alta energía que sí provocan la aparición de pares electrón-hueco dando lugar a un pulso de corriente. Además de este tipo de radiaciones naturales proveniente de los rayos cósmicos, a nivel terrestre, se pueden encontrar también radiaciones producidas por el hombre (instalaciones nucleares), capaces de inducir fallos transitorios en los sistemas electrónicos.

A pesar de que la ocurrencia de este tipo de fallos a nivel terrestre no es tan común como a nivel aeroespacial, se han presentado numerosos incidentes alrededor de todo el mundo. Por ejemplo, en el año 2000, se informó que debido a fallos inducidos por radiación se presentaron bloqueos en los servidores (equipos de la marca *Sun*) de un número importante de sitios web, incluidos *America Online* y *eBay* [Baumann, 2002]. También, *Hewlett Packard* admitió haber tenido varios problemas en los supercomputadores de *Los Alamos Labs* debido a *fallos transitorios* [Michalak et al., 2005]. Por su parte, *Cypress Semiconductor* ha confirmado que a finales de 2001, uno de sus clientes más importantes reportó un enorme caos en una gran compañía telefónica, y que finalmente se logró identificar que todos los problemas habían sido causados por un fallo transitorio que había provocado un bloqueo en el centro de procesamiento de datos [Ziegler and Puchner, 2004].

Dependiendo de múltiples factores, los impactos de estas partículas de alta energía sobre los componentes electrónicos pueden tener diferentes efectos: podrían causar un efecto no observable, un fallo transitorio en la operación normal del circuito, un cambio en la lógica de estado o un daño permanente en el circuito integrado [Dodd and Massengill, 2003]. A continuación se presenta con mayor detalle la clasificación de los efectos más importantes.

2.2.2. Clasificación de los efectos de la radiación

Los efectos de la radiación en los componentes electrónicos pueden llegar a ocasionar distintos fallos con diferentes consecuencias para el sistema. El impacto de una de estas partículas con un componente electrónico puede producir, directa o indirectamente, ionizaciones en sus estructuras de silicio, afectando a su operación de forma permanente (fallos permanentes) o temporal (fallos transitorios). Los fallos permanentes son los más graves porque implican un daño incorregible en el hardware del sistema. Mientras que los fallos transitorios acontecen físicamente cuando una de estas partículas impacta en un componente electrónico, modificando su comportamiento por un periodo

de tiempo determinado. Esto puede llegar a afectar el comportamiento del sistema completo, al corromper un dato almacenado o la transferencia de una señal [Karnik et al., 2004]. Los fallos transitorios también son conocidos como errores lógicos, o *soft errors* en inglés [Perry et al., 2007].

Existen dos tipos de efectos inducidos por la radiación: los de carácter acumulativo, que pueden ocasionar fallos permanentes a los componentes expuestos por un periodo de tiempo prolongado, como por ejemplo los efectos por dosis total (*Total Ionizing Dose (TID)*); también existen los efectos causados por la acción de una única partícula, que pueden producir tanto fallos permanentes como fallos transitorios. Estos últimos efectos son conocidos en inglés como *Single Event Effect (SEE)*. La Figura 2.3 muestra la clasificación de los efectos más relevantes. A continuación se describe cada uno de los efectos presentados.

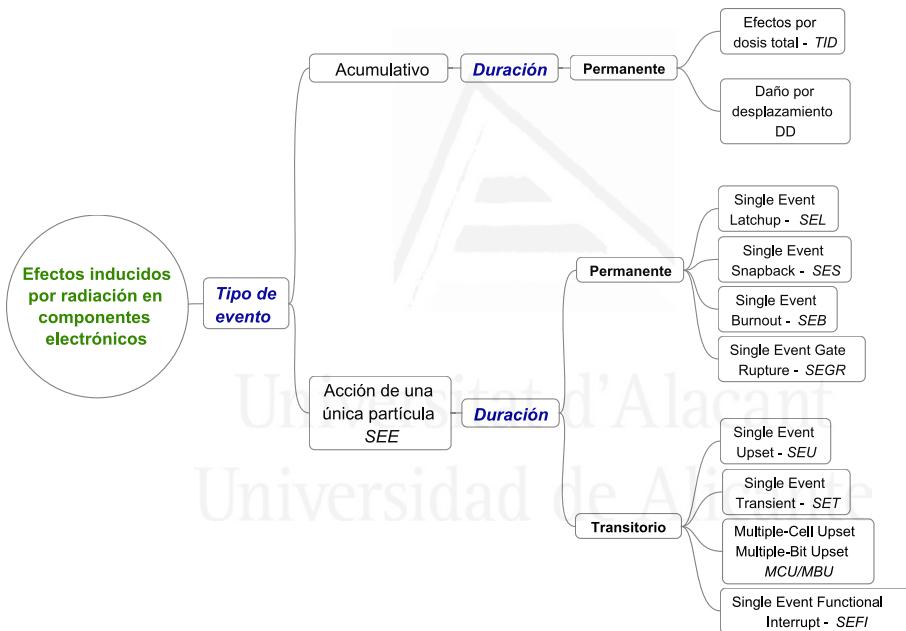


Figura 2.3: Efectos de la radiación en los componentes electrónicos

Efectos de tipo acumulativo

- **Efectos por dosis total (*Total Ionizing Dose - TID*).**

Los efectos por dosis total son el resultado de la exposición a la radiación de forma continua que degrada los componentes y provoca fallos en el funcionamiento del circuito. La radiación genera la aparición de pares electrón-hueco en el óxido de silicio de los transistores *Metal-Oxide-Semiconductor* (MOS). Los huecos tienen una movilidad más baja que los

electrones y mientras que los electrones generados son arrastrados, los huecos quedan atrapados en el óxido, apareciendo una carga neta positiva. Esta carga atrapada modifica el valor de la tensión umbral de puerta del transistor o atrae a electrones hacia la superficie aumentando las corrientes de fuga (*leakage current*). A lo largo del tiempo, estas cargas van aumentando y se acumulan en la interfaz entre el aislante y el silicio, modificando las propiedades del semiconductor. La ionización por dosis total puede provocar que el sistema deje de funcionar.

En las tecnologías actuales, con dimensiones cada vez menores, se ha incrementado la resistencia frente a dosis total ya que el volumen en el que la carga se origina es menor [Pouponnot, 2005, Schrimpf, 2007] y por lo tanto la cantidad de carga generada también es menor. Sin embargo, la sensibilidad frente a los SEEs está en aumento.

- **Daño por Desplazamiento (*Displacement Damage - DD*).**

Consiste en una degradación de las propiedades del material semiconductor debido a que se degenera su estructura cristalina. Al degenerarse la estructura cristalina, se crean puntos locales estables de recombinación, que se traducen en nuevos estados en el diagrama de bandas, situados en la banda prohibida (entre la banda de conducción y la de valencia). Esto se produce como resultado del intercambio de energía cinética con las partículas energéticas que impactan en el dispositivo. Los efectos dañinos provocados por estos intercambios de energía son proporcionales a la sección eficaz de daño por desplazamiento. Esta cantidad es equivalente a la pérdida de energía no ionizante (*NIEL* o *Non Ionizing Energy Loss*). Esta cantidad es responsable del desplazamiento de los átomos en la red cristalina. Se puede encontrar una revisión exhaustiva de toda la literatura relativa a este tipo de fallo en [Srour et al., 2003].

Efectos de Evento Único (*Single Event Effect - SEE*)

La Figura 2.4 representa el efecto causado por la acción de una sola partícula en un transistor. Cuando una partícula con alta energía, impacta un transistor MOS se generan cargas a lo largo de su camino en forma de electrones y huecos. La fuente y el drenador recogen las cargas generadas dando lugar a un pulso de corriente. La carga depositada da lugar a distintos efectos. Cuando el efecto produce un fallo permanente, se denomina error físico o *hard error*; mientras que los efectos que no dañan permanentemente el circuito se denominan errores lógicos o *soft errors*.

A continuación se describen los **errores físicos** más importantes causados por la acción de una única partícula sobre el circuito.

- **Single Event Latch-up (SEL).**

Este tipo de fallo puede ocurrir en tecnología *Complementary Metal-Oxide-Semiconductor* (CMOS) con substrato. Un SEL sucede cuando la carga generada por la partícula ionizante activa transistores bipolares parásitos,

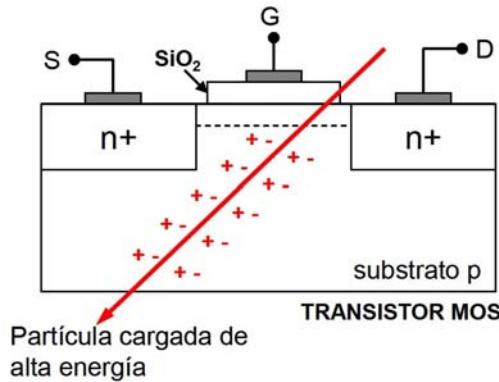


Figura 2.4: Efecto de la acción de una única partícula en un transistor MOS

que se forman entre el sustrato y las diferentes zonas dopadas de los transistores, abriendo un camino entre la alimentación y la tierra, y provocando un cortocircuito. Para eliminar este fallo es necesario interrumpir la alimentación del circuito. Este efecto es muy peligroso puesto que las grandes corrientes que se generan pueden dañar los componentes del circuito permanentemente y dejarlo inservible.

Los fallos SEL pueden evitarse utilizando tecnología *Silicon on Insulator* (SOI), puesto que de esta forma se eliminan los transistores parásitos. Otra forma de evitar estos fallos es aplicar técnicas de *layout* para aumentar la resistencia de los diseños CMOS a este efecto. Generalmente en las aplicaciones espaciales de misión crítica se utilizan tecnologías resistentes a SEL [Pouponnot, 2005].

■ **Single Event Snapping (SES).**

Este fallo es similar al SEL, pero no requiere de la estructura PNPN (tecnología CMOS), ya que puede ser inducido en transistores MOS de canal N que comuten grandes corrientes. Esto ocurre cuando el transistor BJT parásito de tres capas (NPN) se activa y empieza a conducir. La activación de este dispositivo parásito, en el caso de que sea dada por los efectos de la radiación, ocurre mediante la deposición de carga (por parte de una partícula ionizante) en un área sensible del dispositivo [Koga and Kolasinski, 1989]. De igual forma que el efecto anterior, el SES puede observarse de forma macroscópica ya que se pueden medir corrientes extraordinariamente elevadas entre la fuente y el drenador.

■ **Single Event Burnout (SEB).**

El fallo tipo SEB puede ocurrir en transistores de potencia tipo *Metal-Oxide-Semiconductor Field Effect Transistor* (MOSFET). Cuando la unión fuente-substrato se polariza en directa y la tensión drenador-fuente es

mayor que la tensión de ruptura de las estructuras parásitas. La alta corriente resultante y el sobrecalentamiento local pueden destruir el dispositivo [Stassinopoulos et al., 1992].

- ***Single Event Gate Rupture (SEGR).***

La ruptura de compuerta por evento único (SEGR) también se puede observar en los MOSFETs de potencia cuando un ión pesado impacta en la región de puerta mientras una tensión alta es aplicada a la puerta. Una ruptura de dieléctrico local ocurre entonces en la capa aislante de dióxido de silicio, causando sobrecalentamiento y destrucción de la región de puerta [Sexton et al., 1997]. Es importante mencionar que este efecto puede suceder incluso en celdas de almacenamiento *EEPROM* durante la escritura o el borrado, mientras las puertas de los transistores de las celdas están sujetas a una tensión alta.

A continuación se presentan los **errores lógicos**, es decir, los que no causan un daño permanente al circuito, sino que alteran su comportamiento por un periodo de tiempo determinado.

- ***Single Event Upset (SEU).***

Ocurre cuando una partícula impacta en un transistor perteneciente a una celda de almacenamiento (*flip-flop, latch*, o una celda de memoria), y la carga generada es comparable con la carga crítica (carga necesaria para que el transistor commute), provocando el cambio del valor lógico almacenado. En otras palabras, la carga creada por la partícula ionizante se recoge por la propia estructura del biestable. Si la carga total recolectada supera un umbral conocido como carga crítica, el estado lógico del biestable cambia.

Es un fallo transitorio que se modela invirtiendo el valor original del bit afectado durante un ciclo de reloj. El elemento de memoria almacena el valor modificado tras el SEU hasta que se procede a escribir un nuevo valor. De esta forma, el contenido del biestable cambia (de '0' a '1' o bien de '1' a '0').

Este efecto afecta tanto a tecnologías bipolares como a tecnologías CMOS. A medida que se van reduciendo los tamaños de los transistores en tecnologías modernas, también se reduce la carga crítica necesaria para producir un SEU, lo cual agrava el problema, ya que cada nueva tecnología es sensible a partículas ionizantes con menor energía, resultando en que se presenten SEUs a alturas cada vez más cercanas al nivel del mar.

- ***Single Event Transient (SET).***

Un SET se produce cuando una partícula impacta en un transistor perteneciente a lógica combinacional, generando carga que origina un pulso de tensión erróneo de corta duración. La duración de este pulso está en el orden de los 100ps [Pouponnot, 2005]. En tecnologías nanométricas la

duración de la transición es comparable al retardo de propagación de la puerta, por lo que el error se puede propagar a través del circuito y llegar a las salidas o almacenarse en un elemento de memoria o varios. Estos efectos transitorios están cobrando más interés con el aumento de las frecuencias de funcionamiento en los circuitos actuales.

En caso que un SET sea capturado por un solo biestable, se produce un efecto equivalente al de un SEU. La probabilidad de que esto ocurra depende de la frecuencia del reloj, del ancho del pulso transitorio y también del estado de la lógica (dependiendo de este estado el pulso se propagará o resultará enmascarado).

Por otra parte, si un SET se propaga, es posible que sea capturado finalmente por varios biestables, es decir, tendría el efecto de múltiples SEUs. Este efecto se conoce con el nombre de *Single Event Multiple Upsets* (SEMU).

■ ***Multiple Cell Upset (MCU) / Multiple Bit Upset (MBU).***

Son múltiples fallos transitorios producidos como consecuencia de la acción de una sola partícula ionizante que alcanza un lugar del circuito con una alta densidad de transistores. Debido a que el aumento de la capacidad de integración es cada vez mayor, la densidad de transistores en un circuito integrado ha aumentado considerablemente, lo cual está causando también que aumente la probabilidad de que la carga generada por una sola partícula llegue a afectar a varios biestables a la vez. La ocurrencia de este fallo depende de la topología del circuito, ya que deben encontrarse varios biestables en un área muy pequeña, como es el caso de memorias *Random-Access Memory* (RAM), que es donde más suelen ocurrir esta clase de fallos. Mientras más moderna la tecnología, mayor densidad de transistores, y mayor será la probabilidad de que se presenten este tipo de fallos.

Cuando el fallo múltiple se produce en múltiples celdas de almacenamiento y afecta a varios bits de una misma palabra, se denomina MBU; mientras que si las celdas afectadas no pertenecen a la misma palabra de memoria, el efecto que se conoce como MCU [JEDEC, 2009].

No se deben confundir los SEMUs con los MCU/MBU. A diferencia de estos efectos múltiples, en el caso del SEMU los biestables no tienen que estar cerca los unos de los otros, sólo debe cumplirse que el retraso de propagación del pulso transitorio a los biestables sea similar, de forma que cuando llega el flanco de reloj se captura el transitorio en varios biestables a la vez.

■ ***Single Event Functional Interrupt (SEFI).***

Un SEFI se produce cuando un fallo transitorio provoca la inversión del valor de un bit en un registro crítico del sistema de control, produciendo

una interrupción del funcionamiento. Por ejemplo, cuando un error lógico afecta a la memoria de configuración de una *Field Programmable Gate Array* (FPGA), a la lógica de *reset* de un dispositivo, entre otros.

El mecanismo de generación de estos fallos es el mismo que para los SEU, pero el resultado es distinto porque mientras que un SEU puede no tener un efecto final en la operación del circuito, un SEFI implica por definición un mal funcionamiento que sólo puede corregirse reini-ciando la aplicación. En el ejemplo anterior de la FPGA, sería necesario reprogramar el dispositivo de nuevo para recuperar su funcionamiento [Baumann, 2005a].

La presente tesis se centra en el estudio, tolerancia y reparación de los **errores lógicos** (*fallos transitorios*). Principalmente se abordarán los efectos descriptos por los SEUs, ya que son los más frecuentes debido a las características de las tecnologías actuales. No obstante, es importante mencionar que con el incremento en las frecuencias de funcionamiento se prevé una tasa creciente de SETs, por lo que estos efectos también están cobrando gran importancia en la actualidad, y deberán considerarse en los trabajos futuros que surjan a partir de esta investigación.

2.3. Modelos de fallos

Los modelos de fallos se utilizan para representar de forma simplificada las consecuencias de los distintos fenómenos físicos complejos producidos por los fallos. Para el estudio de circuitos electrónicos resulta adecuado modelar los efectos de los fallos mediante parámetros circuitales, como por ejemplo: corrientes, tensiones y cargas introducidas. Sin embargo, idealmente es conveniente modelarlos de forma digital, ya que esto hace posible abstraer y simplificar el estudio, y además, permite aprovechar las capacidades de los simuladores de lógica digital y los emuladores de fallos hardware existentes.

En un sistema basado en microprocesador los fallos se pueden modelar de dos formas: a nivel de los componentes hardware que implementan el sistema (como por ejemplo, subsistemas de memoria, banco de registros, *Arithmetic Logic Unit* (ALU), registros internos del *pipeline*) o a nivel de sistema, que está relacionado con la información que manipula el sistema (como por ejemplo, los datos y las instrucciones del programa). La Figura 2.5 ilustra la clasificación de los modelos para un sistema basado en procesador [Goloubeva et al., 2006a].

2.3.1. Modelos de fallos a nivel de los componentes hardware

Según la duración de los fallos y sus características, existen diferentes modelos de fallos que se pueden utilizar. Principalmente se agrupan en dos categorías: los modelos que sirven para representar fallos permanentes y los modelos para representar fallos transitorios [Pflanz, 2002].

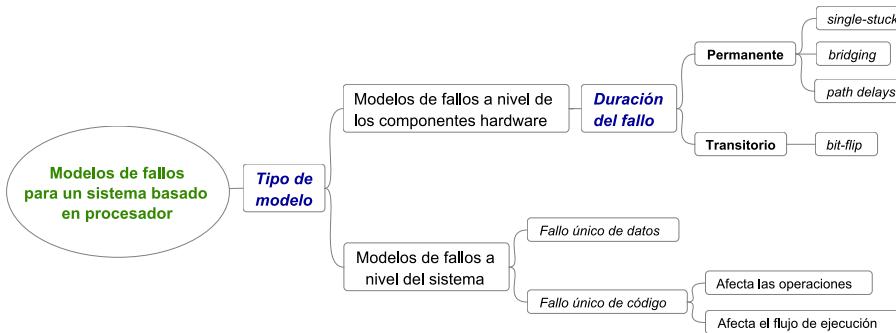


Figura 2.5: Clasificación de los modelos de fallos para un sistema basado en procesador

Modelos de fallos permanentes

Estos modelos son útiles para representar distintos fallos físicos que perturban a un sólo elemento del sistema de forma permanente.

- **Modelo *single-stuck*.**

Este modelo es uno de los modelos de fallo clásicos. Consiste en que el valor de la salida de un componente (afectado por el fallo) debe permanecer fijo de forma permanente en un valor lógico ('0' o '1'), independientemente de los valores que éste tenga en su entrada [Abramovici et al., 1990].

- **Modelo *bridging*.**

Consiste en que dos señales estén conectadas cuando no deberían estarlo (puenteadas). Usualmente se restringe a señales que sean adyacentes físicamente en el diseño [Storey and Maly, 1990].

- **Modelo *path delays*.**

Este modelo requiere de análisis temporal. La señal eventualmente asume el valor correcto, pero con un tiempo de retardo mayor al de la operación normal del circuito. En algunos casos poco frecuentes, incluso es posible que la señal tome el valor correcto un tiempo menor al habitual [Vierhaus et al., 1993].

Modelos de fallos transitorios

Este modelo sirve para representar distintos fallos de tipo lógico que perturban a un sólo elemento del sistema de forma transitoria.

- **Modelo *bit-flip*.**

Consiste en la alteración del estado lógico de un único elemento de memoria en el sistema. Cuando una celda de almacenamiento (*flip-flop*, celda

de memoria, *latch*) es afectado por un *bit-flip*, se realiza una inversión del valor lógico almacenado, es decir, su valor cambia de '0' a '1' o viceversa.

El modelo *bit-flip* es un modelo digital y, como tal, es independiente de la tecnología. Ésto resulta especialmente útil para los diseñadores, ya que se pueden simular/emular *bit-flips* en los distintos niveles de abstracción en los que podemos describir un circuito.

A pesar de que los fenómenos físicos no son instantáneos sino que tienen lugar durante un intervalo de tiempo, el *bit-flip* digital ocurre en un instante de tiempo único y determinado.

Este modelo de fallo es de gran importancia para el estudio de fallos transitorios inducidos por radiación (ver la Sección 2.3.3).

2.3.2. Modelos de fallos a nivel del sistema

Al considerar la información que manipula el sistema, se pueden identificar los siguientes modelos de fallo [Goloubeva et al., 2006a]:

Modelo de fallo único de datos

Está definido como un fallo lógico que afecta a los datos almacenados del sistema. Nótese que la definición no especifica la ubicación concreta donde están almacenados los datos físicamente; podrían estar almacenados en el subsistema principal de memoria, en la cache de datos del procesador, o en el banco de registros del procesador.

Modelo de fallo único de código

Se define como un fallo lógico único que afecta a una instrucción del código fuente del programa. Como en el caso anterior, no se considera donde está ubicada la instrucción errónea; podría estar en la memoria de programa, el subsistema principal de memoria, en la cache de instrucciones del procesador, o en el *pipeline* del procesador.

A su vez, este modelo de fallos sirve para representar dos tipos de fallos diferentes, asociados a un fallo en el código fuente:

- Un fallo que **afecta la operación que la instrucción ejecuta**, pero no afecta al flujo de ejecución del programa. Por ejemplo, cuando un fallo en el código de operación de la instrucción, sustituye la instrucción ADD por una instrucción SUB; o cuando el tipo de direccionamiento de un registro se cambia de directo a indirecto.
 - Un fallo que **modifica el flujo de ejecución del programa**. Por ejemplo, cuando la dirección de destino de un salto se altera, o cuando la condición de un salto condicional se modifica.
-

Nótese que los fallos modelados desde el punto de vista del sistema (fallos de datos o de código) en realidad son abstracciones de fallos que ocurren a nivel de los componentes de hardware.

2.3.3. Modelado de fallos transitorios inducidos por radiación mediante el modelo *bit-flip*

El modelo de fallo *bit-flip* resulta de especial utilidad para representar los efectos de la radiación en los componentes electrónicos, específicamente, aquellos fallos transitorios ocasionados por la acción de una única partícula ionizante. A continuación se describe la forma como pueden representarse dichos efectos utilizando el modelo *bit-flip*:

- **Fallo SEU:** un fallo SEU se puede modelar mediante un *bit-flip* en una única celda de almacenamiento del circuito. El elemento de memoria almacena el valor modificado tras el SEU hasta que se procede a escribir un nuevo valor.
- **Fallo SET:** un fallo SET se modela mediante un único *bit-flip* en un elemento de la lógica combinacional del circuito. Si este SET se propaga hasta varios elementos de la lógica secuencial y se registra en ellos, se pueden considerar como un fallo SEMU. Requiere análisis temporal.
- **Fallo MCU/MBU:** se puede representar un fallo de este tipo mediante múltiples *bit-flips* en varios biestables cercanos unos de otros. Requiere del análisis previo de la topología del circuito.
- **Fallo SEFI:** de la misma forma que para modelar un fallo SEU. Únicamente varían las consecuencias que el fallo ocasiona en el sistema.

A pesar de su simplicidad, el modelo de fallo *bit-flip* es ampliamente usado y aceptado por la comunidad científica de los efectos de la radiación en los componentes electrónicos, ya que este modelo se aproxima a las consecuencias del comportamiento real de los fallos transitorios causados por la acción de una única partícula ionizante [Rebaudengo et al., 2001].

En el contexto de esta tesis doctoral, se utilizará el modelo de fallo *bit-flip* para representar el efecto de los fallos transitorios inducidos por radiación, específicamente, se usará para modelar fallos tipo SEU.

2.4. Medios para alcanzar la confiabilidad

Las principales estrategias para alcanzar la confiabilidad de sistemas electrónicos pueden agruparse en cuatro grupos según [Avizienis et al., 2004]:

- **Prevención de fallos:** consiste en realizar tareas de preparación y disposición de forma anticipada para evitar los fallos. Por ejemplo, durante el

desarrollo de un sistema hardware/software, la prevención de fallos debe ser uno de los principales objetivos. Las mejoras en los procesos de desarrollo ayudarán a reducir el número de fallos del sistema.

- **Eliminación de fallos:** consiste en reducir el número de fallos y su nivel de severidad. Puede realizarse durante la fase de desarrollo del sistema o cuando éste está en uso. Durante el desarrollo la eliminación de fallos consiste en tres pasos: verificación, diagnóstico y corrección. Por otra parte, la eliminación durante la fase de uso del sistema consiste en realizar mantenimientos correctivos o preventivos.
- **Previsión de fallos:** consiste en estimar el número de fallos actual y su posible incidencia en el futuro del sistema, y las consecuencias que traerán consigo.
- **Tolerancia a fallos:** se refiere a la capacidad que tiene un sistema o componente para continuar su operación normal a pesar de que se produzcan fallos en el *hardware* o el *software* [IEEE, 2000]. Existen técnicas de detección de fallos y de recuperación de fallos.

A diferencia de las demás estrategias que están más enfocadas al proceso de desarrollo del sistema, la tolerancia a fallos se centra en la correcta operación del sistema. Y teniendo en cuenta, además, que los fallos transitorios inducidos por radiación son fenómenos físicos que ocurren cuando el sistema está operando normalmente, los diseñadores recurren a las técnicas de tolerancia a fallos para mitigar estos efectos en aquellos sistemas electrónicos donde la confiabilidad sea una de las prioridades.

Las técnicas de tolerancia a fallos pueden consistir en soluciones tecnológicas o en técnicas basadas en diseño [Lacoe et al., 2000].

2.4.1. Soluciones tecnológicas

Las soluciones tecnológicas consisten en la modificación del proceso de fabricación con el objetivo de que el circuito fabricado sea menos sensible a los efectos provocados por la radiación. Esto permite disminuir la tasa de fallos e incluso eliminar por completo algunos de los efectos [IBM, 2011, Colinge, 2001]. Existe una gran variedad de propuestas en la actualidad, que van desde aquellas que implican sustituir algunos de los materiales para la fabricación de los circuitos integrados [Baumann, 2005b, Baumann et al., 1995], hasta enfoques que proponen modificar los procesos de fabricación de los circuitos electrónicos [Roche et al., 2004]. Además, existen tecnologías orientadas a ser robustas frente a radiaciones que se utilizan en las partes críticas de las aplicaciones aeroespaciales. Estas tecnologías, denominadas *radiation-hard* o *rad-hard*, garantizan la tolerancia a fallos por dosis total y una menor sensibilidad a fallos SEL que son los efectos que pueden dar lugar a fallos permanentes o a la destrucción del circuito, y aumentando la resistencia frente a SEUs [Xilinx, 2011],

Altera, 2011, Actel, 2011]. La tecnología *rad-hard* requiere un proceso de fabricación especial mucho más caro que para otra tecnología por lo que es usada únicamente en aplicaciones muy específicas.

2.4.2. Soluciones basadas en el diseño del sistema

Las técnicas de tolerancia a fallos basadas en el diseño del sistema son ampliamente aceptadas ya que pueden ser aplicadas en diferentes niveles del diseño sin necesidad de realizar ninguna modificación en los procesos de fabricación. Existen un gran número de propuestas de este tipo [Nicolaidis, 2005, Calin et al., 1996]. Este trabajo de investigación se centra en esta clase de técnicas.

Las técnicas basadas en el diseño se basan en agregar al sistema alguna funcionalidad adicional, cuyo único objetivo es evitar que algún fallo que ocurra en el sistema se convierta finalmente en un error. El término para referirse a estas funcionalidades adicionales se conoce como *redundancia*. La redundancia puede aplicarse a: el tiempo, la información, los recursos físicos, etc.; más allá de lo que se requeriría normalmente para la operación del sistema [Pradhan, 1996]. El proceso de inserción de estructuras tolerantes a fallos en el diseño del sistema se conoce como *endurecimiento* (del inglés *hardening*).

Los cuatro tipos de redundancia diferentes son:

- Redundancia hardware
- Redundancia de información
- Redundancia temporal
- Redundancia de software

Redundancia hardware

Se llama redundancia hardware a la replicación física de los componentes hardware de un sistema. Estos componentes realizan la misma tarea de tal forma que si un módulo falla, es posible detectar o incluso corregir el fallo. Dependiendo de la acción que se tome cuando se produzca un fallo las técnicas se denominan pasivas, activas o híbridas.

- **Redundancia Pasiva:** se basa en un mecanismo de voto para enmascarar la ocurrencia de un error en el sistema. La Figura 2.6, representa el concepto de redundancia pasiva, donde los resultados de tres versiones idénticas del sistema se conectan a un módulo de votación por mayoría. Este módulo decide la salida del sistema con base en las respuestas entregadas por los tres módulos idénticos, si uno de los módulos ha fallado, el votador por mayoría es capaz de identificar cual es la salida correcta, en base en la respuesta de los otros dos módulos libres de fallos. Este concepto básico es conocido como Triple Redundancia Modular (*Triple Modular Redundancy - TMR*) [Von-Neumann, 1956]. La redundancia pasiva se usa

por lo general para dar tolerancia a los fallos: dado que el votador los enmascara, y nunca alcanzan la salida del sistema.

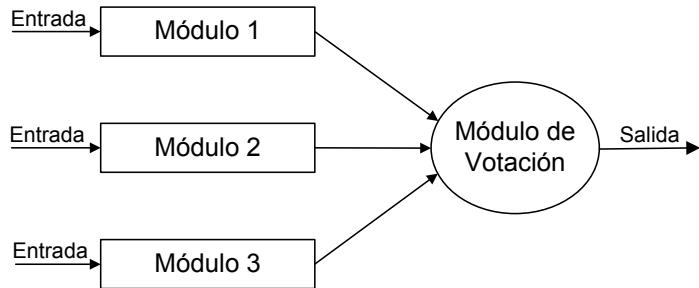


Figura 2.6: Concepto de Triple Redundancia Modular – *TMR*

- **Redundancia Activa:** divide el problema de tolerar los fallos en tres fases: detección, ubicación, y recuperación. La principal diferencia con el tipo de redundancia anterior es que la redundancia activa no intenta enmascarar los errores. Esto implica que la salida del sistema puede ser errónea mientras el sistema trata de detectar, localizar y corregir el error. Un ejemplo de redundancia activa, es la aproximación conocida como *standby sparing* [Avizienis, 1976]. El sistema está compuesto por un módulo operativo y por uno o más módulos extra. Tan pronto como se detecta y localiza un fallo en el módulo operativo, sin importar qué técnica de detección se use, el módulo operativo se reemplaza por uno de los módulos extra. La fase cambio entre el módulo operativo dañado y el módulo extra implementa la fase de recuperación necesaria para restaurar la operación normal del sistema. La entrada de los módulos extra puede ser en frío o en caliente. La presencia de módulos extra en frío significa que dichos módulos están en espera (*idle*) y que aquel que deba entrar en funcionamiento será alimentado sólo cuando deba reemplazar al módulo operativo dañado. Durante la fase de cambio, se interrumpe momentáneamente el servicio prestado por el sistema. Si se quiere disminuir el tiempo de duración de fase de cambio, se puede implementar la presencia módulos extra en caliente. Esta opción implica que los módulos extra se encuentran encendidos y funcionando en paralelo junto al módulo operativo. Tan pronto como se detecta un fallo en el módulo operativo, cualquiera de los módulos extra lo puede reemplazar inmediatamente.
- **Redundancia híbrida:** combina los dos tipos descritos anteriormente. Se implementa el enmascaramiento de los fallos para evitar que alcancen la salida del sistema, mientras se detecta, localiza y recupera el módulo afectado y se restaura el funcionamiento del sistema a un estado libre de fallos.

Redundancia de información

La redundancia de información consiste en la adición de información a los datos para permitir la detección de fallos, enmascarándolos y posiblemente tolerándolos [Pradhan, 1996]. La redundancia de información se basa en la representación de un dato usando un conjunto de reglas autodefinidas. Una porción de los datos es representada en base a las reglas de un código, esta porción se denomina palabra de código. La parte de datos será válida si cumple con todas las reglas que el código define, o será no válida, si viola una o más reglas de éste. La operación de codificación convierte una porción de datos en una palabra de código. Por el contrario, la operación de decodificación convierte una palabra de código en su correspondiente porción de datos.

Dependiendo de las reglas definidas, es posible clasificarlos en:

- **Códigos de detección de error (Error Detection Code - EDC):** permiten detectar la ocurrencia de un error construyendo una palabra de código tal que cualquier error que la modifique la transforme invariablemente en una palabra de código no válida.
- **Códigos de corrección de error (Error Correction Code - ECC):** partiendo de la palabra de código no válida, permiten identificar la correspondiente palabra de código válida.

Como ejemplo, podemos considerar el código de paridad de un solo bit (*single-bit parity code*). Este código impone la adición de un bit extra a un dato binario de tal modo que la palabra de código resultante tenga un número par de *unos* (*paridad par*) o un número impar de *unos* (*paridad impar*). Si una palabra de código de paridad impar presenta un número par de *unos*, esto indica que está siendo afectada por un error que ha cambiado el valor de uno de sus bits, por tanto su paridad se ha convertido en par. En este caso, la detección de errores se hace simplemente contando cuantos *unos* hay en la palabra de código. Sin embargo, existen mecanismos de códigos EDAC mucho más complejos.

Redundancia temporal

Se trata de la repetición, dos o más veces de la misma operación, comparando los resultados de cada intento, con el fin de detectar la ocurrencia de un error.

Cuando se detecta un error, se puede repetir la operación para verificar si el error persiste o ha desaparecido. Se puede clasificar en dos versiones de redundancia temporal:

- **Redundancia temporal para detectar errores transitorios:** se usa para detectar en el sistema la presencia de errores que afectan el correcto funcionamiento del sistema durante un periodo finito de tiempo. La Figura 2.7 presenta cómo el mismo cálculo se repite dos veces, uno en el instante T_0 y el otro en $T_0 + \Delta$. Los resultados de ambos cálculos se comparan, en caso de diferencias entre ambos se señala la detección de un error. La

efectividad de esta técnica se basa en la capacidad del diseñador para identificar un intervalo Δ entre los dos cálculos, de tal modo que sólo uno de los resultados sea erróneo (si es el caso).

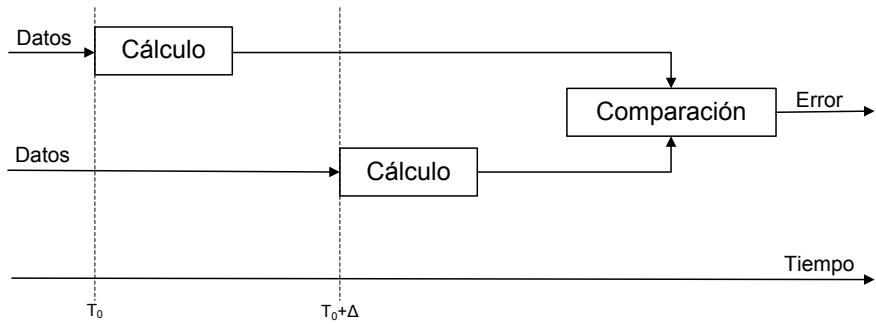


Figura 2.7: Redundancia temporal para detectar errores transitorios

- **Redundancia temporal para detectar errores permanentes:** es una extensión de la anterior, modificada para detectar errores que afecten el sistema durante períodos infinitos de tiempo. La Figura 2.8 presenta el concepto básico. Se hace el primer cálculo en el modo usual. Antes de la realización del segundo cálculo, los datos de entrada se codifican, son sometidos al cálculo y finalmente los resultados son decodificados y comparados con los obtenidos durante el cálculo inicial. Las operaciones de codificación y decodificación se eligen de tal modo que permitan la detección de errores permanentes. Los operadores típicos son los desplazamientos y los complementos [Pradhan, 1996].

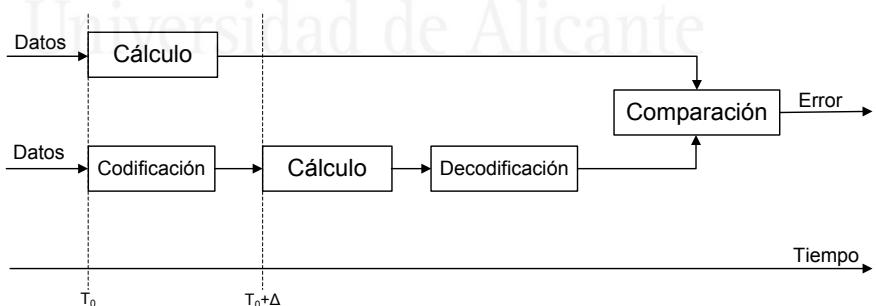


Figura 2.8: Redundancia temporal para detectar errores permanentes

Redundancia de software

En aplicaciones que usan microprocesadores, se puede utilizar la redundancia software. Esta técnica proporciona flexibilidad y no requiere ninguna

modificación hardware, por lo que se puede usar para detectar y corregir fallos permanentes y transitorios con un coste muy bajo de implementación. Este mecanismo consiste en replicar parte del código o añadir software extra que realice funciones de comprobación, de modo que implica un incremento en los requisitos de memoria y tiempo de ejecución. Los fallos pueden generar errores en los datos o en el flujo de control de la ejecución y existen distintas técnicas orientadas a tratar cada uno de estos problemas, como por ejemplo: [Cheynet et al., 2000, Oh et al., 2002b].

En la siguiente sección se presenta el estado actual de las técnicas de tolerancia a fallos basadas en el diseño del sistema.

2.5. Estado actual de las técnicas de tolerancia a fallos

Los diseñadores pueden usar diferentes estrategias para proporcionar características de tolerancia a fallos a los sistemas que están diseñando. Las técnicas basadas en la redundancia hardware han sido tradicionalmente el enfoque más utilizado para enfrentar problemas de confiabilidad en el diseño de circuitos digitales. No obstante, gracias al auge actual de los sistemas basados en microprocesador, han surgido una gran cantidad de técnicas de protección basadas en la utilización de software redundante; y enfoques híbridos basados en la combinación de protecciones hardware y software. Además, durante los últimos años han surgido múltiples propuestas basadas en mecanismos de protección aplicables a dispositivos de lógica programable (como las FPGAs) [Mitra et al., 2005]. A continuación se describen las principales propuestas publicadas en la literatura acerca de cada una de estas estrategias.

2.5.1. Técnicas de tolerancia a fallos basadas en hardware

Entre los métodos de protección basados en redundancia hardware se pueden encontrar una gran variedad de propuestas. Por un lado, existen una gran variedad de propuestas basadas en la redundancia de información para la protección de memorias. Por otro lado, también existe un amplio repertorio de técnicas basadas en la mitigación de fallos mediante modificaciones en la lógica del circuito, que van desde nivel de compuertas lógicas hasta el nivel de la arquitectura del sistema.

Protección de memorias basada en redundancia de información

Las memorias representan la parte más grande de los diseños modernos. Además, son muy susceptibles a partículas ionizantes ya que están diseñadas para alcanzar la mayor densidad posible. Por lo tanto, la protección de memorias es una de las prioridades más importantes cuando se diseña un sistema tolerante a fallos inducidos por radiación [Nicolaidis, 2011b].

Mientras que los fallos permanentes en memorias pueden solucionarse mediante técnicas *Built-In Self-Repair* (BISR) (como en [Nicolaidis et al., 2003]), esto no es posible cuando se trata de fallos transitorios inducidos por radiación. Para este tipo de fallos, es necesario utilizar técnicas que enmascaren los fallos basadas en la redundancia de información. Por este motivo es necesario utilizar códigos de detección de errores (*Error Detection Code* - EDC) o códigos de corrección de errores (*Error Correction Code* - ECC); en conjunto conocidos como códigos de detección y corrección de errores (*Error Detection and Correction Code* - EDAC).

Algunas de estas técnicas pueden alcanzar un alto grado de fiabilidad. A continuación se mencionan las principales técnicas basadas en redundancia de información:

■ ***Error Detection Codes* (EDCs):**

- Bit de paridad.
- *Adler-32*.
- *Cyclic Redundancy Check* (CRC).
- *Fletcher checksum*.
- Algoritmo *Verhoeff*.
- *Longitudinal Redundancy Check* (LRC).

■ ***Error Correction Codes* (ECCs):**

- Códigos de *Hamming*.
- Códigos *BCH*, por ejemplo: *Berlekamp-Massey*, *Peterson-Gorenstein-Zierler*, *Reed-Solomon*.
- Algoritmo *BCJR*.
- *Forward error correction*.
- Código *Gray*.

Los códigos de corrección de errores son el mecanismo más habitual para reducir la tasa de errores en memorias, en particular en SRAM al ser este tipo de memoria el más sensible a los SEEs. Sin embargo, hay que tener en cuenta que el área necesaria para la implementación de dichas técnicas es mayor para los códigos correctores que para los que únicamente detectan errores, en especial si corrigen errores múltiples. Por tanto, el diseñador debe encontrar un compromiso entre la penalización en área y complejidad del sistema, y la tolerancia a fallos que se obtiene.

En [Neuberger et al., 2005] y [Hentschke et al., 2002] pueden encontrarse implementaciones de los códigos *Reed-Solomon* [Reed and Solomon, 1960] y *Hamming*, respectivamente.

En [Nicolaidis, 2005] y [Nicolaidis, 2003] se proponen soluciones para implementar EDAC en memorias con mecanismos de lectura/escritura no convencionales, ya que esta puede ser una tarea más difícil que para las memorias

convencionales. Por ejemplo, para las memorias que permiten enmascarar las operaciones, ya que algunas escrituras modifican todos los bits de la palabra, pero otras no (sólo un subconjunto de ellos). Cuando esto ocurre, se desconoce el contenido de los bits restantes y por eso no se pueden calcular los bits de chequeo para esta palabra. También, en el mismo trabajo se proponen mecanismos para la protección de memorias tipo *Content-Addressable Memories* (CAM).

Protección de la lógica del circuito

Las propuestas basadas en la utilización de mecanismos de hardware redundante para proteger la lógica del circuito son muy variadas. Existen soluciones a bajo nivel, es decir, a nivel de transistor, que proponen modificaciones de las celdas de memoria; o a nivel más alto, *Register Transfer Level* (RTL) o de sistema, introduciendo estructuras redundantes durante el diseño del circuito. A continuación se resumen los enfoques más relevantes.

A nivel de compuertas lógicas, se pueden proteger de los fallos inducidos por radiación, diferentes clases de circuitos como: microprocesadores, memorias, ASICs, circuitos reprogramables, entre otros; reemplazando sus celdas de almacenamiento tradicionales (celdas SRAM, *flip-flops*, *latches*) por celdas de almacenamiento protegidas (*endurecidas*).

Existen varias propuestas de celdas de memoria endurecidas. Una solución típica consiste en modificar el *layout* de la celda, aumentando el tamaño del transistor o añadiendo capacidades adicionales para conseguir incrementar la carga necesaria para producir un cambio en el funcionamiento del transistor (carga crítica). Otras técnicas se basan en modificar la arquitectura de la celda.

En [Calin et al., 1996] se presenta una estructura para celdas de memoria SRAM, denominada *Dual Interlocked Cell* (DICE), que es tolerante a cualquier fallo transitorio que afecte a un único nodo de la celda. En una celda SRAM tradicional, cuando un SEE afecta a un nodo, el valor almacenado cambia. Sin embargo, en una celda DICE, dos de los nodos siguen almacenando el valor correcto, de modo que en cuanto el efecto del fallo transitorio desaparece, el valor original se restaura. Aplicando esta solución en tecnologías de 90 nm se obtiene una inmunidad total frente a SEUs durante el modo estático de operación, mientras que en el modo dinámico pueden almacenarse estados erróneos si se produce una escritura durante el periodo de recuperación del dato.

En [Gusmão-de Lima, 2000] se puede encontrar un estudio comparativo de diferentes tipos de celdas de almacenamiento endurecidas.

El mayor inconveniente que presentan estos métodos, basados en la modificación de la estructura de la celda para hacerla tolerante a fallos, es el incremento en la cantidad de área necesaria y la disminución de la velocidad. Por ejemplo, para tecnologías de 90 nm, un biestable basado en DICE presenta un 81% de aumento de área con respecto a un biestable industrial [Nicolaidis, 2011b]. No obstante, más recientemente en [Nicolaidis et al., 2008] se han propuesto nuevos principios de diseño de celdas de almacenamiento protegidas, los cuales han sido utilizados por [Lin et al., 2011] para diseñar un nuevo tipo de celda rápida que proporciona la misma protección que DICE, pero requiere 20% me-

nos área que ésta y, además, proporciona el 55 % de reducción del producto entre velocidad y potencia.

Por otra parte, existen una gran variedad de técnicas que se aplican a nivel de sistema o de transferencia de registros (RTL). Consisten en replicar bloques lógicos del circuito para conseguir la tolerancia a fallos. Estas técnicas son fácilmente aplicables por el diseñador en las primeras etapas del ciclo de diseño y con coste más bajo comparado con otro tipo de soluciones.

Los componentes de hardware redundante realizan la misma tarea de tal forma que si un módulo falla es posible detectarlo o corregir el fallo. Se aplican técnicas que van desde la duplicación o triplicación de estructuras de bajo nivel y la adición de votadores por mayoría para detectar/corregir el fallo (por ejemplo: DMR y TMR [Von-Neumann, 1956]); hasta la utilización de componentes mucho más complejos, como por ejemplo:

- El uso de un coprocesador para comparar y validar las salidas del procesador principal [Mahmood and McCluskey, 1988].
- En [Austin, 1999] se propone la arquitectura denominada *Dynamic Implementation Verification Architecture* (DIVA) que utiliza un procesador principal de alto rendimiento que ejecuta las instrucciones, acompañado de una unidad de chequeo funcional que valida la ejecución.
- Modificar los componentes de la microarquitectura de un procesador superescalar para implementar la redundancia [Ray et al., 2001].
- La duplicidad del sistema completo mediante dos FPGAs y comparar las salidas obtenidas en ambos dispositivos [Kubalik and Kubatova, 2008].

La redundancia temporal es una técnica eficaz para detectar o enmascarar fallos transitorios debido al carácter temporal de éstos. Además, las técnicas de redundancia temporal requieren generalmente pocos elementos hardware extra para su implementación y control a diferencia de las soluciones mencionadas antes. Por lo tanto, este tipo de técnicas resultan más adecuadas en circuitos en los que el hardware es un recurso mas crítico que el tiempo, ya sea por el coste, el tamaño o el peso. La redundancia temporal puede implementarse de dos formas distintas:

- Repitiendo los cálculos en varios instantes de tiempo para luego comparar los resultados obtenidos [Johnson, 1989].
- Almacenando los datos en distintos instantes de tiempo sin necesidad de repetir el cálculo [Nicolaïdis, 1999].

Las técnicas basadas en recalcular los datos conllevan una penalización notable en el tiempo de ejecución, pero pueden ser utilizadas para detectar no sólo errores transitorios sino también errores permanentes, añadiendo una cantidad mínima de hardware. Estas técnicas se basan en repetir las operaciones tras haber modificado los operandos de alguna forma (como el concepto presentado en la Figura 2.8). Para ello, se realiza la operación en un instante de

tiempo T_0 y se guardan los resultados obtenidos. A continuación, se codifican los operandos mediante alguna función y se vuelve a realizar la misma operación en el instante de tiempo $T_0 + \Delta$, almacenando los nuevos resultados. Sobre estos últimos se aplica la función inversa para decodificar los resultados y poderlos comparar con los resultados obtenidos inicialmente. Entonces, si no hay fallos, ambos resultados deben coincidir. Dependiendo de la modificación aplicada se distinguen distintas técnicas. Las propuestas existentes más destacadas son las siguientes:

- Repetición de las operaciones con lógica complementaria. Se aplica típicamente a la transmisión de datos, aunque también puede utilizarse en algunos circuitos. Consiste en transmitir en primer lugar el dato original, y posteriormente, su complemento.
- Repetición de las operaciones tras realizar un desplazamiento del operando. Esta técnica es conocida como *REcomputing with Shifted Operands* (RESO) [Patel and Fung, 1982] y sirve para detectar fallos en la unidad aritmética.
- Repetición de las operaciones tras realizar un intercambio entre las partes más y menos significativas del operando. Se conoce como *REcomputing with Swapped Operands* (RESWO).
- Repetición de las operaciones utilizando duplicación con comparación. Esta técnica es conocida como *REcomputing with Duplication with Comparison* (REDWC). Consiste en combinar redundancia temporal con la redundancia hardware. Se divide cada operando en dos partes. Primero se realiza la operación por duplicado (redundancia hardware) de la parte más significativa y se comparan los resultados para detectar posibles errores en la parte más significativa del resultado. Después, se realiza la operación (redundancia temporal) con la parte menos significativa, también por duplicado, obteniendo el resultado final.

Otros autores han propuesto técnicas que están fundamentadas en el aprovechamiento de la multiplicidad de bloques hardware disponibles en arquitecturas multi-hilo/multi-núcleo para implementar la redundancia. En el trabajo de [Saxena and McCluskey, 1998] se habló por primera vez de utilizar hilos redundantes para mitigar los fallos transitorios. Posteriormente, se presenta la técnica *AR-SMT* en [Rotenberg, 1999] que usa el concepto de multi-hilo simultáneo o *Simultaneous Multi-Threading* (SMT). Por otra parte, la técnica conocida como multi-hilo redundante simultáneo o *Redundant Multi-Threading* (RMT), propuesta por [Reinhardt and Mukherjee, 2000], incrementa el rendimiento de *AR-SMT* y compara flujos redundantes antes que los datos se almacenen en memoria para permitir la detección de fallos. Posteriormente, se adiciona recuperación de fallos a RMT en el procesador *Simultaneous and Redundant Multi-Threaded processor with Recovery* (SRTR) [Vijaykumar et al., 2002]. También, en [Mukherjee et al., 2002] se presentan mejoras basadas en RMT y SMT, y se presenta el procesador *Chip-level Redundantly Threaded* (CRT), el cual

ha sido extendido a recuperación de fallos posteriormente en el llamado *Chip-level Redundant Threading with Recovery* (CRTR) [Gomaa et al., 2003].

De forma adicional a todos los enfoques hardware mencionados hasta ahora, más recientemente, algunos autores han propuesto el *endurecimiento selectivo* de algunos módulos del sistema. Por ejemplo, en [Samudrala et al., 2004] se propone proteger únicamente las partes más vulnerables del circuito; en [Zoellin et al., 2008] se seleccionan selectivamente las compuertas lógicas que deben ser endurecidas durante las etapas tempranas del diseño, priorizando la protección de sectores críticos del circuito que pueden ocasionar un malfuncionamiento; otras propuestas pretenden reducir la degradación del rendimiento ocasionada por las protecciones, mediante la aplicación de forma parcial de técnicas de redundancia multi-hilo [Parashar et al., 2006, Reddy et al., 2006]; o como en el trabajo de [Ergin et al., 2009], se propone la utilización de estructuras hardware simples para proteger los valores almacenados (aquellos que almacenan *valores estrechos o narrow values*), replicando únicamente una parte de sus bits (los utilizados), en su parte disponible.

A pesar de que la mayoría de las propuestas basadas en hardware proveen una solución efectiva para la mitigación de fallos transitorios, por lo general estas técnicas conllevan una grave penalización al sistema en términos de recursos utilizados, consumo de potencia, área, tiempo de diseño y costes económicos. Por esta razón, su utilización no es viable en muchos casos, en especial, cuando se trata del diseño de sistemas embebidos que no cuentan con procesadores con grandes prestaciones.

2.5.2. Técnicas de tolerancia a fallos basadas en software

Para los sistemas basados en microprocesadores, se puede utilizar la redundancia software para obtener tolerancia a fallos. Esta estrategia de protección proporciona flexibilidad y no requiere ninguna modificación hardware por lo que se puede usar para detectar y corregir fallos permanentes y transitorios con un coste muy bajo de implementación (es posible utilizar componentes COTS). Por esta razón, durante los últimos años se han desarrollado numerosas propuestas basadas en software redundante, para brindar a los programas capacidades de detección/corrección de fallos, asegurando un nivel de confiabilidad aceptable [Rebaudengo et al., 2011].

Para conseguir que un sistema sea tolerante a fallos es necesario completar satisfactoriamente dos tareas: la detección y la recuperación de los fallos (subsanación de los efectos causados por los fallos). Debido al hecho de que estas dos tareas son fácilmente desacoplables, en la mayoría de los casos, son tratadas de forma independiente. Además, debido a que la ocurrencia de un fallo es un suceso poco frecuente por lo general, el código de subsanación de errores se ejecuta con mucha menor frecuencia que el código de detección, que debe estar ejecutándose continuamente. El diseño de técnicas de detección de fallos eficientes y a bajo coste es, por tanto, uno de los temas fundamentales en la literatura sobre tolerancia a fallos, y de hecho gran parte de las propuestas se ocupan exclusivamente de este aspecto, dejando abierta la posibilidad de ser

complementadas para alcanzar la recuperación de los fallos.

Al considerar los posibles efectos que tienen los fallos que afectan al software de un sistema basado en microprocesador, surgen dos categorías principales: aquellos fallos que afectan el flujo de control de la ejecución del programa, como por ejemplo, al modificar el código de operación de una instrucción justo antes de ejecutarse; o aquellos fallos que cambian los datos con los que trabaja el programa, por ejemplo, al corromper el valor de un resultado. Existen distintas técnicas orientadas a tratar cada uno de estos problemas, como se presenta en las siguientes secciones. Sin embargo, normalmente ambos tipos de técnicas se aplican conjuntamente para cubrir tanto fallos en los datos como en el control de la ejecución.

Técnicas para proteger el flujo de ejecución del programa

En esta sección se presentan las técnicas software más importantes para la protección de los sistemas basados en microprocesador contra los errores de control de flujo, más conocidos por sus siglas en inglés como *Control Flow Errors* (CFEs). Estos errores causan que el procesador ejecute una instrucción diferente a la esperada.

Las técnicas de comprobación de control de flujo se basan en dividir el programa a ejecutar en *bloques básicos* de ejecución. Un *bloque básico* es un conjunto de instrucciones que se ejecutan de forma consecutiva, sin contener ningún salto o llamada a función, excepto quizás en la última instrucción, ni tampoco una instrucción que sea el destino de una llamada o salto, excepto en el caso de la primera instrucción.

De esta forma, un programa se representa mediante un grafo de control de flujo o *Control Flow Graph* (CFG). El cual está formado por un conjunto de nodos, que son cada uno de los *bloques básicos* de ejecución; y un conjunto de relaciones entre ellos (aristas), donde cada una de ellas representa un cambio en el flujo de ejecución del programa (saltos, llamadas y retornos de subrutinas). Por ejemplo, en la Figura 2.9 se muestra un grafo de control de flujo, donde I_i representa una instrucción del código fuente del programa.

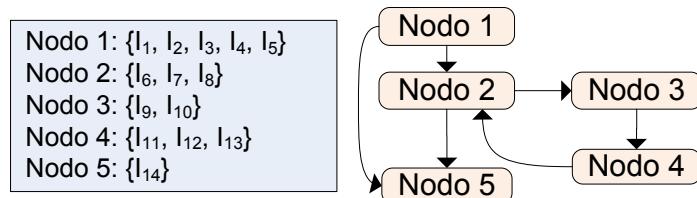


Figura 2.9: Grafo de control de flujo de un programa

La base de las técnicas software de este tipo consiste en identificar el grafo de control de flujo del programa e identificar cada nodo de manera única al inicio de su ejecución con una firma. Cuando se finaliza la ejecución de las instrucciones pertenecientes al nodo (*bloque básico*), se comprueba que el identificador

es correcto. De esta forma, en caso que tras la comprobación, el identificador no tenga el valor esperado, se detectaría un fallo. Este tipo de estrategias son conocidas como *técnicas de monitorización de firmas*.

Los tipos de fallos que se pueden encontrar en el flujo de ejecución de un programa son los siguientes:

- Un fallo causa un salto ilegal desde el final de un bloque básico hasta el inicio de otro bloque básico.
- Un fallo ocasiona un salto legal pero equivocado desde el final de un bloque básico hasta el inicio de otro bloque básico.
- Un fallo produce un salto desde el final de un bloque básico hasta cualquier punto de otro bloque básico.
- Un fallo causa un salto desde cualquier punto de un bloque básico hasta cualquier punto de otro bloque básico.
- Un fallo ocasiona un salto desde cualquier punto de un bloque básico hasta cualquier otro punto del mismo bloque básico.

Nótese que los primeros cuatro tipos de fallo suceden mediante saltos entre bloques básicos diferentes, mientras que el último ocurre dentro del mismo bloque básico.

Son muchas las propuestas publicadas de técnicas software de tolerancia a fallos de este tipo. En [Goloubeva et al., 2006b] se hace un completo estudio al respecto. Se diferencian principalmente porque algunas requieren una mayor o menor cantidad de recursos (tamaño del programa, tiempo de ejecución, entre otros factores). También se diferencian de acuerdo a cuales de los tipos de fallos mencionados son capaces de detectar. Sin embargo, en este trabajo sólo se mencionan las técnicas más representativas:

- En [Benso et al., 2001] se propone una técnica de monitorización de firmas donde el chequeo del flujo de ejecución de los programas se implementa mediante el aprovechamiento de las características de multiproceso/multi-hilo del sistema operativo.
 - En [Venkatasubramanian et al., 2003] se propone la técnica conocida como *Assertions for Control Flow Checking* (ACFC). Consiste en asignar un bit de una variable especial a cada uno de los bloques básicos (esta variable se llama *estado de ejecución*). Se adiciona código al programa de forma tal que se ponga a *uno* el bit de la variable de estado cuando se ejecute el bloque básico correspondiente. Cuando termina el programa, se compara la variable de estado con una constante, cuyo valor es *uno* en todos los bits que representan los bloques básicos. El valor de la variable de estado debe coincidir con la constante en caso que no haya habido ningún fallo.
 - En [Goloubeva et al., 2003] se presenta la técnica llamada *Yet Another Control flow Checking Approach* (YACCA). A cada bloque básico le son asignados dos identificadores únicos, uno para su entrada y otro para su salida.
-

Se introduce dentro del programa un código que se va calculando mediante una operación que involucra los identificadores únicos de cada bloque. Es posible detectar errores, al verificar que el código tiene un valor distinto al esperado. En [Goloubeva et al., 2005] se proponen algunas mejoras a esta técnica.

- En la técnica conocida como *Control-flow Error Detection through Assertions* (CEDA) [Vemu and Abraham, 2006] también son asignados dos identificadores únicos a cada nodo. Un registro contiene el valor de la firma, la cual es continuamente actualizada para monitorizar la ejecución del programa. La firma se actualiza a la entrada y a la salida de cada nodo. Se hace la comprobación de la firma en ciertos puntos del programa llamados *puntos de chequeo*. En [Vemu and Abraham, 2008] se proponen mejoras a CEDA con el fin de reducir la memoria utilizada y la degradación del rendimiento.
- *Control-Flow Checking by Software Signatures* (CFCSS) [Oh et al., 2002b]. Es una de las técnicas más conocidas y utilizadas. En este caso se insertan instrucciones nuevas dentro del programa para controlar los flujos entre bloques básicos. Las firmas (*Signatures*) se incluyen en el programa durante la compilación, usando un campo constante de las instrucciones y posteriormente, dichas firmas, son generadas en tiempo de ejecución y comparadas con las almacenadas durante la compilación. Ambas deben coincidir, indicando que no ha existido ningún fallo.

Técnicas para proteger los datos

La aproximación clásica a estos métodos de tolerancia a fallos es conocida como *N-versions programming* [Avizienis, 1985]. Consiste en la replicación N veces del software y el enmascaramiento del error mediante la obtención de un resultado válido por votación de entre todas las réplicas (*majority voting*). Esta técnica produce un incremento de $100(N - 1)\%$ en área y tiempo de computación. Las técnicas que se revisan a continuación pueden considerarse refinamientos de ésta, que persiguen reducir el impacto de la redundancia sobre dichos parámetros.

La redundancia software puede aplicarse con diferentes niveles de granularidad:

- Programa.
- Procedimiento.
- Instrucción.

Los métodos basados en la *redundancia software a nivel de programa* pueden estar fundamentados en tres estrategias distintas:

1. Redundancia temporal.

Como se mencionó en la Sección 2.4.2, cuando se adopta la redundancia temporal, el mismo cálculo se lleva a cabo en instantes de tiempo diferentes usando el mismo hardware. Un caso particular de este tipo de redundancia consiste en ejecutar dos programas con la misma tarea y los mismos datos de entrada, pero en instantes de tiempo distintos. En este caso, sólo es necesario un procesador, que ejecute el mismo programa dos veces o dos programas en secuencia que implementen el mismo algoritmo. Posteriormente, al final de la ejecución del último programa se deben comparar los resultados obtenidos para comprobar que no hay errores. Esta técnica puede ser extendida para detectar errores permanentes (además de los transitorios) al hacer que la segunda instancia del programa sea diferente, aunque semánticamente siga siendo equivalente. Un ejemplo de esta técnica se puede encontrar en [Jochim, 2002].

2. Ejecución simultánea.

Este tipo de técnicas aprovechan la multiplicidad de bloques hardware existente en procesadores multi-hilo/multi-núcleo para implementar la redundancia, y de esta forma, ejecutar varias instancias del programa de forma simultánea. Ver la sección 2.5.1.

3. Diversidad de datos.

Consiste en ejecutar dos programas que tengan la misma funcionalidad, pero que alcancen el objetivo del programa mediante conjuntos de datos diferentes. Al finalizar la ejecución de los programas se comparan los resultados para detectar fallos transitorios o permanentes.

Para aplicar esta técnica se parte de un programa original, y se genera una nueva versión de éste, al multiplicar todas las variables y constantes por un factor de diversidad k . Dependiendo del factor de diversidad, la versión original y la versión transformada del programa pueden usar diferentes partes del hardware, y por consiguiente, propagar los fallos de diferentes formas. La Figura 2.10 presenta un ejemplo básico de este concepto; se muestra el código fuente de un programa y su versión diversificada al aplicarle un factor de diversidad $k = -2$. Un ejemplo de este tipo de técnica es conocido como *Error Detection by Diverse Data and Duplicated Instructions* (ED⁴I) [Oh et al., 2002a].

Por otro lado, la *redundancia software a nivel de procedimiento* consiste en efectuar la duplicación de la llamada a los procedimientos de forma selectiva. Esta técnica es mejor conocida como *Selective Procedure Call Duplication* (SPCD) [Oh and McCluskey, 2002]. Cada procedimiento (o un subconjunto de ellos) es llamado dos veces con los mismos parámetros; el resultado de cada una de las ejecuciones se almacena y se compara, permitiendo de esta manera la detección de errores. La Figura 2.11 muestra un ejemplo de un programa donde el procedimiento *B* se llama dos veces.

Versión original	Versión transformada
<pre>x = 1; y = 5; i = 0; while (i < 5) { z = x + i * y; i = i + 1; } i = 2 * z;</pre>	<pre>x = -2; y = -10; i = 0; while (i > -10) { z = x + i * y / (-2); i = i + (-2); } i = (-4) * z / (-2);</pre>

Figura 2.10: Ejemplo de un programa en su versión original y trasformada mediante diversidad de datos

```
int a, a1, b, c;

void A2()
{
    a = B(b);
    a1 = B(b);
    if (a <> a1)
        error();
    c = c + a;
}

int B(int b)
{
    int d;
    d = 2 * b;
    return(d);
}
```

Figura 2.11: Ejemplo de un programa con redundancia software a nivel de procedimiento

Las técnicas de *redundancia software a nivel de instrucción* se fundamentan en la idea de que la operación que realiza una instrucción del código fuente del programa puede ser replicada, y los resultados obtenidos pueden ser comparados para detectar/corregir errores. En el caso de que se ejecute una instrucción duplicada justo después de la instrucción original, ésta puede realizar exactamente la misma operación que la instrucción original, o puede llevar a cabo una operación modificada de la instrucción original.

Existen propuestas basadas en la redundancia de instrucciones partiendo del código fuente escrito en un lenguaje de alto nivel (principalmente lenguaje C) y otras basadas en la redundancia de instrucciones a bajo nivel (código ensamblador) con el fin de reducir el impacto en el tamaño de código y el tiempo

de ejecución del programa tras el endurecimiento, y también, para mejorar la tasa de detección de fallos.

En [Rebaudengo et al., 1999] se propone un método llamado *Automatic Rule Based Transformation* (ARBТ) que está basado en la redundancia de instrucciones a alto nivel (lenguaje ANSI C) para la detección de errores. El enfoque principal de la técnica se basa en la duplicación de código y datos de acuerdo con una serie de reglas de transformación aplicadas directamente al código fuente. La gran ventaja de este método radica en el hecho de las reglas de transformación pueden ser incluso aplicadas de forma automática al código fuente, como en [Rebaudengo et al., 2001]. Esto libera al programador de la carga de tener que garantizar la integridad y efectividad de la técnica (por ejemplo, al elegir qué duplicar y dónde hacer el chequeo). Además, el método es completamente independiente del hardware sobre el cual se trabaja y es posible que complemente otros mecanismos de detección de error ya existentes. Esta técnica se busca detectar los errores que afectan al código y los datos, mediante la duplicación de cada variable y la adición de chequeos de consistencia después de cada operación de lectura. La Figura 2.12 presenta un ejemplo de la aplicación de esta técnica.

Código original	Código modificado
<pre>int a,b; a = b;</pre>	<pre>int a₀, b₀, a₁, b₁; a₀ = b₀; a₁ = b₁; if (b₀ != b₁) error();</pre>
<pre>a = b + c;</pre>	<pre>a₀ = b₀ + c₀; a₁ = b₁ + c₁; if ((b₀ != b₁) (c₀ != c₁)) error();</pre>

Figura 2.12: Ejemplo de aplicación de reglas para detectar errores en los datos

En los trabajos de [Benso et al., 1998, Benso et al., 2000, Cheynet et al., 2000, Rebaudengo et al., 2000] se puede encontrar más información relacionada a ARBT, como por ejemplo: detalles acerca de la implementación, herramientas de inyección de fallos utilizadas para la evaluación de la confiabilidad, y los resultados obtenidos para esta técnica. Además, el mismo grupo de investigadores ha publicado estudios acerca de la efectividad de las técnicas software en la mitigación de los efectos de los SEUs y SETs [Rebaudengo et al., 2002a, Lisboa et al., 2006].

En [Nicolescu et al., 2004] se realiza un análisis de distintas clases de fallos que escapan a la detección por las técnicas software tradicionales (técnicas de duplicación de instrucciones y técnicas de monitorización de firmas). Se proponen técnicas específicas para detectar algunas clases especiales de fallos, que podrían complementar estos trabajos.

La propuesta inicial de ARBT se concibió inicialmente sólo para la detección de errores, dejando sin especificar el procedimiento externo que se ocuparía de la recuperación del error. No obstante, posteriormente en los trabajos de [Rebaudengo et al., 2002b, Rebaudengo et al., 2003, Rebaudengo et al., 2004] se proponen una serie de mejoras y extensiones de las reglas para contemplar la tolerancia a fallos, aunque sólo para el caso de SEUs que afecten al segmento de datos de la memoria. Esta técnica se conoce como *Automatic Ruled Based Transformation for Fault Tolerance* (ARBT-FT).

Por otra parte, existen propuestas basadas en la redundancia de instrucciones a bajo nivel (código ensamblador) con el fin de reducir el impacto en el tamaño de código y el tiempo de ejecución de los programas. En [Oh et al., 2002c] se presenta la técnica conocida como *Error Detection by Duplicated Instructions* (EDDI), en la cual se mejora la tasa de detección de fallos y se reduce el impacto en el rendimiento considerablemente, con respecto a otras técnicas de detección de fallos basadas en redundancia de instrucciones a alto nivel. EDDI es un método que busca aprovechar el *Instruction Level Parallelism* (ILP) entre las instrucciones originales y las duplicadas al ejecutarse sobre arquitecturas superescalares. En la Figura 2.13 puede verse un ejemplo de un programa endurecido con esta técnica.

I_1 : ADD R1, R2, R3	I_1 : ADD R1, R2, R3
I_2 : SUB R4, R1, R2	I_2 : SUB R4, R1, R7
I_3 : AND R5, R1, R2	I'_1 : ADD R21, R22, R23
I_4 : MUL R6, R4, R5	I_3 : AND R5, R1, R2
I_5 : ST R6	I'_2 : SUB R24, R21, R27
	I_4 : MUL R6, R4, R5
	I'_3 : AND R25, R21, R22
	I'_4 : MUL R26, R24, R25
	I_c : BNE R6, R26, ir_gestor_error
	I_5 : ST R6
	I'_5 : ST R26

Figura 2.13: Ejemplo EDDI

En los trabajos de [Hu et al., 2009, Hu et al., 2005] se propone un algoritmo para balancear los efectos negativos que implica la confiabilidad sobre el rendimiento y se plantea una forma de mantener un nivel de desempeño constante variando el nivel de confiabilidad requerido.

Además, en el trabajo de [Reis et al., 2005b] se expone una técnica de detección de fallos en los datos y en el flujo de ejecución. Esta técnica es conocida como *SoftWare Implemented Fault Tolerance* (SWIFT) y se plantea como una técnica con bajo impacto sobre el tamaño del código y el rendimiento del programa, ya que aprovecha, como en el caso de EDDI, el ILP entre instrucciones originales y duplicadas. En este caso se implementa sobre arquitecturas *Very Long Instruction Word* (VLIW) que ofrecen más oportunidades para colocar las

operaciones duplicadas en los *slots* libres de las instrucciones VLIW. SWIFT es una técnica basada en un conjunto de transformaciones llevadas a cabo durante la compilación, las cuales duplican las instrucciones del programa e insertan instrucciones de comparación en puntos estratégicos. Durante la ejecución del programa, los valores son calculados dos veces y comparados para detectar fallos transitorios. Esta propuesta se basa en la técnica EDDI + CFCSS, planteando algunas variaciones para mejorar los resultados de confiabilidad. La Figura 2.14 muestra un ejemplo de un programa sencillo endurecido con esta técnica.

<pre> ld r3 = [r4] add r1 = r2, r3 st[r1] = r2 </pre>	<pre> 1: br faultDet, r4 != r4' ld r3 = [r4] 2: mov r3' = r3 add r1 = r2, r3 3: add r1' = r2', r3' 4: br faultDet, r1 != r1' 5: br faultDet, r2 != r2' st[r1] = r2 </pre>
---	---

Figura 2.14: Ejemplo programa protegido con la técnica *SWIFT*

De igual forma, también han surgido otras propuestas que buscan aprovechar la redundancia intrínseca en las arquitecturas VLIW. Se pueden encontrar más ejemplos de este tipo en [Bolchini, 2003, Bolchini and Salice, 2001].

Posteriormente, los mismos autores de SWIFT comentan la ausencia de propuestas concretas para la recuperación de errores dentro de las técnicas software basadas en redundancia de instrucciones a bajo nivel. En [Reis et al., 2007] proponen tres técnicas distintas, que conllevan distintos niveles de cobertura de fallos y degradación del rendimiento.

- **SoftWare Implemented Fault Tolerance Recovery (SWIFT-R):** es un método basado en la triple redundancia modular (TMR). En SWIFT se utiliza la doble redundancia modular, lo cual proporciona detección pero no recuperación de errores. En SWIFT-R en lugar de crear una copia redundante como en SWIFT, se crean dos copias redundantes, teniendo en total tres copias (con la original), mediante las cuales en caso de existir algún error (tipo SEU) se hacen comparaciones de las tres versiones y mediante la inserción de votadores por mayoría. De esta forma, se identifica cual es la copia que está corrupta y se restaura el valor correcto.
- **TRUMP:** Doble redundancia con protección por multiplicadores. Se basa en la teoría de códigos-AN (AN-Codes), los cuales permiten representar la redundancia de una forma más compacta. TRUMP contiene los datos redundantes de SWIFT-R en dos registros en vez de tres. Sin embargo, la codificación-AN utilizada en TRUMP, es menos general que TMR usada en SWIFT-R, ocasionando que no sea posible proteger ciertas partes de los

programas. En *TRUMP* se obtiene un mejor rendimiento que en *SWIFT-R* ya que no se deben triplicar las instrucciones.

- **Mask:** es una técnica estática de bajo coste para el enmascaramiento de errores. Se identifican de antemano posibles errores, y se intentan resolver de forma estática dentro del código del programa.

Asimismo, durante los últimos años se han desarrollado enfoques que están basadas en optimizaciones automáticas realizadas a los programas en tiempo de compilación. Estas optimizaciones buscan reducir la tasa de *soft errors*. Se pueden encontrar algunos ejemplos de estos enfoques en los trabajos de [Lee and Shrivastava, 2009a, Lee and Shrivastava, 2009b, Liem et al., 2008].

A pesar de que las técnicas basadas en software redundante son más efectivas en cuanto al coste en comparación con las técnicas hardware, la redundancia software ocasiona dos importantes inconvenientes: incremento de las necesidades de memoria y degradación del rendimiento [Azambuja et al., 2010a]. La primera circunstancia puede limitar gravemente la aplicabilidad de la propuesta, ya que se deben considerar las restricciones típicas de los sistemas embebidos y de los procesadores en los que están basados (sobre todo si son de 8 bits). El segundo inconveniente tiene una importancia relativa, dependiendo del tipo de aplicación que se desee endurecer. Sin embargo, merece la pena mencionar que algunas de estas técnicas software de tolerancia a fallos han sido utilizadas exitosamente para la mitigación de los efectos de la radiación en sistemas reales de misión crítica, como subsistemas de satélites y otros que participan en diferentes misiones espaciales [Pignol, 2010, Shirvani et al., 2000].

2.5.3. Técnicas híbridas de tolerancia a fallos

Para obtener confiabilidad en un sistema basado en microprocesador, no sólo existen las técnicas de tolerancia a fallos con enfoques puramente software o hardware, también existen los enfoques híbridos (hardware/software). La estrategia principal de estos últimos consiste en combinar la redundancia software con algún tipo de soporte externo basado en hardware.

Como se ha expuesto en las dos secciones anteriores, existen una gran variedad de métodos de tolerancia a fallos basados en hardware y otros basados en software. Sin embargo, cada uno de estos tipos de redundancia tiene asociado algunos inconvenientes. Las técnicas hardware conllevan una grave penalización al sistema en términos de recursos utilizados, consumo de potencia, área, tiempo de diseño y costes económicos. Por otra parte, las técnicas software causan el incremento de las necesidades de memoria y la degradación del rendimiento de los programas. Por estos motivos, la aplicabilidad de estas propuestas no resulta viable en muchos casos, principalmente cuando se trata del diseño de sistemas embebidos de bajo coste que no cuentan con procesadores con grandes prestaciones. No obstante, en muchas ocasiones la solución óptima es un punto intermedio entre las técnicas hardware y las técnicas software, es decir, una técnica híbrida que combine estrategias de protección software y hardware. De esta forma, se puede obtener lo mejor de ambas y encontrar un

balance adecuado entre: coste del sistema, rendimiento, área y nivel de confiabilidad.

Este tipo de enfoques está dirigido principalmente a grandes dominios de aplicación de los sistemas embebidos, donde hoy por hoy, factores como el consumo de potencia, el coste y el rendimiento del sistema; son tan importantes como la confiabilidad del mismo. En este sentido, algunos trabajos recientes en este campo han mostrado resultados prometedores.

En el trabajo de [Reis et al., 2005c, Reis et al., 2005a] se propone una técnica híbrida para la detección de fallos. Esta técnica es denominada *CompileR Assisted Fault Tolerance* (CRAFT). La técnica está basada en la técnica software de detección de fallos SWIFT [Reis et al., 2005b], pero en este caso es extendida mediante estructuras hardware presentadas en la técnica hardware RMT [Mukherjee et al., 2002]. La técnica tiene tres variaciones. La primera de ellas combina SWIFT con una estructura hardware que protege los almacenamientos de los datos en memoria. La segunda, combina SWIFT con un mecanismo hardware que sirve para duplicar y proteger los datos cargados desde la memoria. Por último, la tercera variación consiste en combinar SWIFT con las dos estructuras hardware mencionadas antes para proteger los almacenamientos y las cargas.

Por otro lado, en [Bernardi et al., 2006a, Bernardi et al., 2006b] se presenta una técnica híbrida para la detección de fallos. Este método está basado en las reglas de transformación del código fuente de para proteger los datos de ARBT [Cheynet et al., 2000] y las reglas para proteger el control de flujo de ejecución de YACCA [Goloubeva et al., 2003]. Esto se logra mediante el soporte externo de un mecanismo hardware externo que está conectado al bus del sistema, y permite aliviar el esfuerzo computacional del procesador. De esta forma, es posible reducir el impacto en el rendimiento del programa y mejorar la capacidad de detección de fallos. Recientemente esta técnica ha sido extendida a detección y recuperación de fallos en [Bernardi et al., 2010, Berriardi et al., 2007], cuya estrategia se fundamenta en las reglas de trasformación del código fuente propuestas en [Rebaudengo et al., 2004].

En [Azambuja et al., 2011, Azambuja et al., 2010b] se propone una técnica híbrida para la detección de SEUs y SETs. Está basada en transformaciones al código que implementan un algoritmo de monitorización de firmas. Además, es soportada por un módulo hardware externo (*watchdog + decodificador*) que permite detectar fallos de control de flujo y verificar el flujo de datos y direcciones entre el procesador y la memoria principal. En este sentido, también en [Li and Gaudiot, 2010] se presenta un enfoque similar, donde además se aprovechan los conceptos de SMT para alcanzar la recuperación de los fallos mediante la ejecución redundante del mismo hilo.

Otras propuestas presentan una arquitectura para procesadores superescalares que es soportada por métodos de tolerancia a fallos basados en software para ofrecer la posibilidad de detectar y corregir errores con muy poco impacto en el rendimiento del sistema [Scholzel, 2010]. Además, recientemente en [Lee and Shrivastava, 2010] se ha propuesto realizar la protección parcial del banco de registros para encontrar un balance apropiado entre el nivel de con-

fiabilidad y la energía del sistema.

A pesar de las aportaciones recientes, las soluciones híbridas existentes todavía son muy específicas y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre las restricciones de diseño y los requisitos de confiabilidad.

2.5.4. Técnicas de tolerancia a fallos para FPGA

Los circuitos actuales tienden a la integración de distintos componentes en un mismo chip, *System on Chip* (SoC), proporcionando altas prestaciones con altas densidades de integración. De esta forma, los diseños actuales, cada vez más complejos, presentan nuevos retos que requieren el desarrollo y propuesta de nuevas técnicas y soluciones para el diseño tolerante a fallos. En particular, las FPGAs proporcionan una solución interesante para una gran cantidad de aplicaciones por sus prestaciones y coste. En esta sección se presentan los avances más recientes para el diseño de sistemas tolerantes a fallos inducidos por radiación para FPGAs.

Los fabricantes de FPGAs ofrecen tres alternativas principales para este segmento de mercado: las FPGAs basadas en memoria SRAM, las FPGAs basadas en memoria *Flash*, y las FPGAs antifusibles.

Las FPGAs basadas en SRAM son dispositivos utilizados para cada vez más aplicaciones, gracias a sus altas prestaciones y a que pueden ser re-programados tantas veces como sea necesario modificando la memoria de configuración (que es una memoria SRAM). Sin embargo, este tipo de FPGAs es muy sensible a los efectos de la radiación. Un SEU en la memoria de configuración puede modificar las funciones lógicas implementadas en el dispositivo o las interconexiones (produciendo cortocircuitos, circuitos abiertos o asignaciones erróneas de señales), o interrumpiendo el funcionamiento normal (SEFI). Los bits de la memoria de configuración suponen generalmente más de un 95 % del número total de bits susceptibles de sufrir un SEU en una FPGA. Por lo tanto, resulta necesario proteger frente a SEE's tanto el diseño, como la memoria de configuración.

En cuanto a las FPGAs basadas en memoria *Flash*, cuentan también con la gran ventaja de ser re-programables, pese a que están basadas en un tipo de memoria no-volátil. Aunque las celdas *Flash* son inmunes a radiación de iones pesados, pueden ser necesarios esquemas de tolerancia frente a los efectos de la radiación dependiendo de la criticidad de la aplicación.

Por último, las FPGAs antifusibles sólo se pueden programar una vez. Requieren de un proceso de fabricación especial. La gran ventaja que tiene este tipo de FPGAs es que son inmunes frente a SEE's, aunque en casos extremos pueden ser necesarias técnicas de mitigación de fallos.

Los esquemas de mitigación de los efectos de la radiación en FPGAs incluyen los siguientes:

- Aplicar soluciones relacionadas con la tecnología y la arquitectura interna, es decir, mediante el desarrollo de FPGAs fabricadas con elementos

tolerantes a radiación. Esta solución conlleva la fabricación de un nuevo chip endurecido (conocidos como *rad-hard FPGAs*). Por lo general, este tipo de FPGAs es muy costoso, y además ofrecen menos prestaciones que las comerciales. Las tecnologías con las que se fabrican proporcionan menor densidad de integración y no son reconfigurables, eliminando completamente los efectos SEUs de la memoria de configuración (FPGAs antifusible). Su uso se limita a aplicaciones de misión crítica cuyo presupuesto sea muy elevado (por ejemplo, en sistemas espaciales). Algunos ejemplos de FPGAs *rad-hard* son: *Actel RTAX FPGAs* [Actel, 2010], y *Xilinx Virtex-5QV* [Xilinx, 2010].

- Existen soluciones a nivel de sistema, basadas en la utilización de dispositivos redundantes que utilizan una doble o triple FPGA con mecanismos de votación por mayoría para validar las salidas del sistema, como por ejemplo en [Kubalik and Kubatova, 2008].
- Aplicar técnicas de endurecimiento por diseño a alto nivel, introduciendo redundancia en el circuito al instrumentar el código de descripción de hardware implementado (*Hardware Description Language (HDL)*). Estas técnicas permiten utilizar FPGAs comerciales con altas prestaciones y un bajo coste. Pueden aplicarse para endurecer la lógica de usuario, como los registros, las memorias empotradas, multiplexores, entre otros [Kastensmidt et al., 2006]. Su implementación se puede llevar a cabo de dos formas:
 - Instrumentando el diseño de forma manual. Aunque se ofrece total flexibilidad al diseñador, esta forma suele ser costosa en tiempo y además, propensa a errores.
 - Instrumentando el diseño de forma automática. Es posible utilizar herramientas automáticas para aplicar mecanismos básicos de endurecimiento. Algunas de estas herramientas son: *Mentor Precision Rad-Tolerant* [Mentor, 2010], *Synopsys Synplify Pro* [Synopsys, 2010] y *Xilinx TMR Tool (XTMR)* [Xilinx, 2009].
- Propuestas recientes están fundamentadas en realizar el proceso de distribución física del diseño en la FPGA (*place and route*) de forma orientada a mejorar la confiabilidad. Estas técnicas garantizan que, por ejemplo, en el caso de un diseño que esté protegido mediante TMR, ningún fallo SEU causará un fallo múltiple tal que invalide el esquema de protección de TMR. Algunos ejemplos de estas técnicas se pueden encontrar en [Sternone and Violante, 2006] y [Huang et al., 2011].
- Además, para resolver el problema de los fallos en la memoria de configuración de las FPGAs basadas en memoria SRAM es necesario utilizar otras soluciones como la reconfiguración periódica del dispositivo, más conocida como *scrubbing*. Existen dos variaciones de esta técnica:

- Configuración *bitstream scrubbing*: reconfigurar la FPGA de forma constante a una frecuencia más alta que la tasa de fallos predicha o esperada [Xilinx, 2000].
- Configuración *bitstream repair*: se hace un *read-back* del *bitstream*, se calcula un CRC en el *bitstream* para detectar fallos, y se realiza una reconfiguración parcial para corregir errores. Esta variación también es conocida como *scrubbing avanzado* [Kastensmidt et al., 2006].

En las propuestas de [Kastensmidt et al., 2006], [Kastensmidt et al., 2004], [Lima et al., 2003], y [Lima et al., 2000] se pueden encontrar una gran cantidad de propuestas y enfoques para alcanzar la tolerancia a fallos en FPGAs basadas en memorias SRAM. Asimismo, en los trabajos de [Cassel and Lima, 2006] y [Serpone et al., 2007] se evalúan varias técnicas para este tipo de FPGAs. La discusión de estos trabajos está por fuera del ámbito de esta tesis, sin embargo, se mencionan en este apartado para guiar al lector.

2.6. Evaluación de la confiabilidad

Una de las tareas más importantes a la hora de diseñar un sistema tolerante a fallos consiste en tener la posibilidad de evaluar la confiabilidad del sistema que se está diseñando. Para ello es necesario disponer de métricas y parámetros específicos que permitan valorar la tolerancia a fallos de un sistema. Estos parámetros pueden ser cualitativos o cuantitativos. Mientras que los parámetros cualitativos describen de forma subjetiva las bondades de un sistema, los parámetros cuantitativos son medidas objetivas representadas con un valor numérico que permiten obtener información acerca de aspectos como: saber si un sistema alcanza los niveles de confiabilidad mínimos exigidos, comparar la confiabilidad de distintos sistemas entre sí, y evaluar el efecto de aplicar técnicas de endurecimiento en un diseño. En esta sección se mencionan las técnicas existentes para evaluar la confiabilidad, y además se presentan los parámetros y métricas que serán utilizados en este trabajo para cuantificarla.

2.6.1. Técnicas para la evaluación de la confiabilidad

Existen distintos métodos para realizar la evaluación del nivel de confiabilidad de un sistema. Principalmente se pueden clasificar en métodos analíticos y métodos experimentales [Benso and Prinetto, 2003].

Los métodos analíticos tienen como objetivo obtener un valor numérico de las propiedades de la confiabilidad (fiabilidad, disponibilidad, seguridad, etc.), utilizando modelos matemáticos basados en teorías estadísticas y de probabilidad, como por ejemplo *modelos de Markov*, *redes de Petri estocásticas* o *modelos combinatorios*. Los métodos analíticos permiten caracterizar los mecanismos de tolerancia a fallos de forma exacta y predecir su comportamiento frente a fallos cuando el modelo desarrollado es preciso. Sin embargo, la representación matemática de circuitos reales es una tarea de gran dificultad y los modelos que

se obtienen suelen ser demasiado complejos para realizar los análisis. Cuando se aplican simplificaciones en dicho modelado se pierde precisión reduciéndose la validez de los resultados obtenidos. Además, en el caso de utilizar *IPs* (*Intellectual Property*), lo cual es muy habitual en los sistemas modernos, la estructura interna del circuito es desconocida y no se dispone de la información necesaria para desarrollar un modelo matemático del circuito.

Los métodos experimentales se basan en medir directamente los parámetros de confiabilidad del sistema. Esto se puede conseguir mediante dos formas diferentes:

- Observar y registrar el comportamiento del circuito durante su fase de operación estudiando el efecto que provocan los fallos reales. De esta forma, es posible obtener datos como el tiempo medio entre fallos o la probabilidad de suceso de un fallo, aparte de la cobertura de fallos y otras características de los mecanismos de tolerancia a fallos disponibles.
- Introducir fallos en el circuito artificialmente, es decir, *inyectar fallos*, y observar su efecto.

La observación y medida del sistema durante la fase de operación con una carga de trabajo real proporciona información sobre los errores y averías que se producen de forma natural. Esta información incluye datos referentes a la sensibilidad de la tecnología en el entorno real en el que opera el sistema, a la frecuencia con la que se producen los fallos en dicho entorno y al comportamiento real de los mecanismos de tolerancia a fallos de los que dispone el sistema. Por lo tanto, es el método que ofrece los resultados más realistas.

No obstante, la medida de la tolerancia a fallos en la fase de operación no es un método adecuado para predecir el comportamiento del sistema frente a fallos ni para facilitar la inserción de técnicas de redundancia durante su diseño. Se utiliza principalmente para verificar las hipótesis realizadas en los modelos analíticos y validar otros métodos de evaluación de la confiabilidad utilizados en fases anteriores del ciclo de diseño. Para realizar una experiencia en la fase operacional es necesario disponer de un sistema real y utilizar un tiempo de test adecuado. Teniendo en cuenta que la tasa de averías de sistemas críticos tolerantes a fallos puede estar comprendida entre 10^{-5} y 10^{-9} averías/hora, el tiempo necesario para obtener datos estadísticamente relevantes puede ser excesivo (cientos de años).

Para reducir el tiempo de observación necesario, estos experimentos se desarrollan en estaciones situadas a una altitud elevada para que el flujo de partículas con alta energía sea mayor que a nivel terrestre. Otra forma de reducir el tiempo de observación consiste en realizar el test en tiempo real sobre varios dispositivos en lugar de sobre un único circuito de manera que la probabilidad de que ocurra un fallo aumenta. Aún así, este método de medida no es viable para todos los circuitos que se diseñan y fabrican en la actualidad.

La forma de acelerar la medida experimental de la tolerancia a fallos de un sistema es inyectar fallos artificialmente para observar la respuesta del circuito.

Las técnicas de inyección de fallos pueden aplicarse en distintas etapas del ciclo de diseño y tienen los siguientes objetivos:

- Validar las técnicas de tolerancia a fallos implementadas en el sistema.
- Ayudar en el proceso de endurecimiento al diseñador indicando qué partes del circuito no cumplen con las especificaciones requeridas de confiabilidad.
- Prever cuál será el comportamiento del sistema en presencia de fallos.

La inyección de fallos se aplica de forma relativamente rápida y eficaz en circuitos reales y complejos. Por lo tanto, presenta soluciones a los principales inconvenientes de aplicar métodos analíticos o realizar experimentos en campo. Estas características hacen de dichas técnicas un método potente y ampliamente aceptado para la evaluación de la tolerancia a fallos de un circuito.

Según [Entrena et al., 2011], las técnicas hardware de inyección de fallos pueden clasificarse de acuerdo a la forma cómo inyectan los fallos:

- **Inyección física de fallos.** La inyección de fallos reales consiste en someter el circuito bajo estudio a una perturbación externa que produzca fallos, sin necesidad de modificar el comportamiento del circuito a excepción de la propia inserción del fallo. Por lo tanto, son métodos no intrusivos que no introducen ningún aumento en el tiempo de ejecución con respecto a la duración normal de la ejecución del circuito. Como mayores inconvenientes, presentan una observabilidad y controlabilidad limitadas y la necesidad de un equipo de test adicional para realizar la inyección y observar el resultado.

Dentro de estas técnicas se pueden distinguir los siguientes tipos: inyección mediante experimentos de irradiación de los circuitos [JEDEC, 2006, JEDEC, 1996, Buchner et al., 2002, ESA, 2002]; mediante rayos láser sobre los circuitos [Palomo et al., 2010, Miller et al., 2004], o técnicas de inyección de fallos en los conectores (más conocidas como técnicas de inyección a nivel de pin) [Powell et al., 1995].

- **Inyección lógica de fallos.** La inyección física de fallos implica el uso de equipos especiales para la generación de los fallos, que son caros, especialmente para las técnicas de irradiación. En general, son técnicas con limitaciones en el control de las posiciones dónde se inyectan los fallos y en la observabilidad de sus efectos. Para superar estas limitaciones se realiza la inyección de fallos lógicos aprovechando los recursos propios del sistema a estudiar, como por ejemplo:

- La lógica presente en los circuitos para realizar tareas de depuración, disponible en muchos de los circuitos VLSI actuales. La mayoría de los procesadores actuales contienen lógica dedicada para tareas de depuración. Esta lógica, conocida como *OCD* (*On-Chip Debugger*),

permite al diseñador el acceso a los recursos internos a través de herramientas software. Estas capacidades se pueden utilizar para acceder a los contenidos de la memoria y los registros, proporcionando un mecanismo sencillo para la inyección de fallos en circuitos microprocesadores comerciales, sin necesidad de modificar el software implementado [Fidalgo et al., 2006, Peng et al., 2006].

- Funciones software en el caso de sistemas basados en microprocesador. Las técnicas de inyección de SEU en microprocesadores basadas en software consisten en introducir una rutina software para modificar el valor del elemento de memoria en el que se desea inyectar un fallo. Estas técnicas son rápidas porque se ejecutan a la velocidad normal del procesador y no requieren de hardware adicional para realizar la inyección. Sin embargo, presentan ciertas limitaciones. Sólo se pueden inyectar fallos en posiciones accesibles por software. Son difíciles de generalizar a diferentes sistemas debido a que los métodos se desarrollan generalmente en base a las características y las capacidades del sistema al que se aplica.

La inserción de un fallo puede implementarse en tiempo de compilación o durante la ejecución del programa. Las técnicas de inyección en tiempo de compilación requieren la modificación del programa fuente o el código ensamblador cada vez que se realiza una inyección y son más adecuados para fallos permanentes. Por otro lado, las técnicas de inyección en tiempo de ejecución se utilizan para emular fallos transitorios.

Para activar la inserción del fallo es necesario añadir algún mecanismo. Estos pueden ser: temporizadores [Kanawati et al., 1995], rutinas de interrupción [Carreira et al., 1998, Velazco et al., 2000], inserción de código al programa ejecutado donde se inyecta un fallo antes de que se ejecute alguna instrucción.

- Los recursos disponibles para la reconfiguración de los dispositivos lógicos programables, ya que al cambiar la memoria de configuración se pueden inyectar fallos directamente en los elementos de memoria de los diseños prototipados. Este método es ampliamente utilizado para evaluar el efecto de los fallos en las memorias de configuración de las FPGAs. También, con el fin de reducir el tiempo necesario para la inyección de fallos, es común utilizar mecanismos de reconfiguración parcial [Kenterlis et al., 2006, Alderighi et al., 2010, Alderighi et al., 2003].

- **Inyección lógica de fallos mediante emulación del circuito.** Así como las plataformas de prototipado basadas en FPGAs se han vuelto muy populares en los últimos años para la verificación de ASIC, también ha sucedido esto para evaluar los prototipos mediante inyección de fallos. A diferencia del enfoque anterior, en este caso se usan las FPGAs para soportar la inyección de fallos, pero el diseño final es implementado en ASIC. Este enfoque es conocido como inyección de fallos basado
-

en emulación. Actualmente existen numerosos ejemplos de plataformas que han sido desarrolladas para este fin, algunos ejemplos pueden encontrarse en: [Antoni et al., 2003, Aguirre et al., 2004, Civera et al., 2001, Lopez-Ongil et al., 2007].

2.6.2. Métrica para evaluar la confiabilidad

Según [Mukherjee et al., 2003], los bits de una arquitectura se clasifican de formas diferentes, dependiendo si son necesarios en un tiempo determinado para la correcta ejecución de la arquitectura o no. Del mismo modo sucede con los efectos ocasionados en el programa a causa de un fallo inyectado, se clasifican de acuerdo a la clasificación dada al bit que ha sido objeto del fallo. Los fallos se clasifican de la siguiente manera:

- *unnecessary for Architecturally Correct Execution* (unACE): cuando el programa completa su ejecución y a pesar del fallo inyectado logra obtener los resultados esperados. Esta situación puede darse porque el fallo permaneció pasivo en el sistema hasta que finalizó la ejecución y no llegó a convertirse en un error; o bien, porque la técnica de tolerancia a fallos logró recuperar el error causado por el fallo (en el caso de los programas endurecidos).
- *Silent Data Corruption* (SDC): cuando el programa termina su ejecución normalmente pero no obtiene los resultados esperados a causa del fallo inyectado, es decir, la ejecución del programa ha sido funcionalmente diferente a la operación normal del programa.
- *Hang o Bloqueado*: cuando el programa termina su ejecución de forma anormal o permanece iterando en un ciclo infinito a causa del fallo inyectado.

Nótese que SDC y *Hang* son efectos indeseables para el sistema y se pueden clasificar en su conjunto como fallos que han ocurrido en bits necesarios para la correcta ejecución de la arquitectura o *Architecturally Correct Execution* (ACE) *bits*.

Al llevar acabo una campaña de inyección de fallos, se debe evaluar el efecto que cada uno de estos fallos tiene sobre el comportamiento del sistema, y de esta forma, es posible evaluar el nivel de cobertura frente a fallos que ofrece el sistema. En el caso que se haya aplicado alguna técnica de endurecimiento, se puede cuantificar el aporte obtenido en la fiabilidad del sistema.

Es importante recordar que las técnicas de mitigación de fallos necesariamente introducen redundancia en el sistema, y esto provoca dos hechos que es necesario considerar. Por un lado, estas técnicas incrementan el tiempo de ejecución de los programas, y por consiguiente, la probabilidad de ocurrencia de un fallo también es más alta, porque el sistema estará expuesto a fallos durante un periodo de tiempo mayor que en la versión no-endurecida. Por otra parte, la redundancia hace que se incremente el número de bits activos en el sistema,

y de esta forma, se incrementan también el número de bits propensos a fallos. Por esta razón, al considerar la confiabilidad proporcionada por las técnicas de endurecimiento, no sólo se deben considerar el porcentaje de los fallos injectados que no causan ningún efecto negativo al sistema (porcentaje de fallos unACE), sino que también se debe considerar la degradación del rendimiento que provocan estas técnicas.

La métrica utilizada en este trabajo para medir la confiabilidad del sistema es conocida como la cantidad de *trabajo medio hasta la avería* o *Mean Work To Failure* (MWTF), que ha sido propuesta por [Reis et al., 2005a]. Se trata de una generalización de la métrica conocida como *tiempo medio hasta la avería* o *Mean Time To Failure* (MTTF). La métrica MWTF captura la solución de compromiso que hay entre el nivel de fiabilidad (porcentaje de fallos unACE) y el rendimiento del sistema. El MWTF puede ser calculado de la siguiente forma:

$$MWTF = (\text{tasa de error} \times AVF \times \text{tiempo de ejecución})^{-1} \quad (2.1)$$

donde:

- la *tasa de error* está determinada por el tipo de tecnología del circuito;
- el *tiempo de ejecución* es el tiempo necesario para ejecutar una unidad de trabajo dada (en este caso la unidad de trabajo es la ejecución del programa completo);
- y el *Architectural Vulnerability Factor* (AVF) es la probabilidad que tiene un fallo que ocurre en una estructura hardware determinada de que se convierta en un error [Mukherjee et al., 2003]. El factor de vulnerabilidad de la arquitectura es otra métrica de confiabilidad usada normalmente y se puede calcular de la siguiente manera:

$$AVF = \frac{\text{número de bits ACE en la estructura}}{\text{número total de bits en la estructura}} \quad (2.2)$$

2.7. Conclusiones

En este capítulo se han introducido los conceptos básicos relacionados con la confiabilidad en los sistemas electrónicos que sirven como punto de partida para entender las discusiones planteadas en esta tesis.

Existen diferentes factores ambientales que pueden ocasionar fallos transitorios y alteraciones en el comportamiento de componentes electrónicos, como por ejemplo: las variaciones en la temperatura, las variaciones en la tensión de alimentación, las interferencias electromagnéticas, y los efectos inducidos por radiación. En esta investigación se abordan éstos últimos. Por este motivo, en este capítulo se han presentado los efectos de la radiación en los sistemas electrónicos modernos, lo cual constituye la causa del problema que se quiere abordar en esta investigación.

El problema de los efectos de la radiación en los sistemas electrónicos ya no es exclusivo únicamente de las aplicaciones aerospaciales, ya que cada vez se dan más efectos de este tipo a nivel atmosférico y terrestre. Cada vez más, las aplicaciones de los sistemas electrónicos modernos requieren mecanismos para la mitigación de estos fallos. Por esto, la protección de los sistemas electrónicos contra este tipo de fallos se está convirtiendo en un requisito obligatorio para un número creciente de dominios de aplicación. Entre ellos: espacial, aviónica, automoción, defensa, medicina, nuclear, industria, comunicaciones, etc.

La presente tesis se centra en el estudio, tolerancia y reparación de los **errores lógicos** (*fallos transitorios*). Principalmente se abordarán los efectos descritos por los SEUs ya que son los más frecuentes debido a las características de las tecnologías actuales. No obstante, merece la pena mencionar que con el incremento en las frecuencias de funcionamiento se prevé una tasa creciente de SETs, por lo que estos efectos también están cobrando gran importancia en la actualidad, y serán tenidos en cuenta en esta investigación.

También en este capítulo, se han descrito los principales modelos de fallos utilizados para representar los diferentes y complejos efectos físicos de los fallos. En el contexto de esta tesis doctoral, se utilizará el modelo de fallo *bit-flip* para representar el efecto de los fallos transitorios inducidos por radiación.

Posteriormente en este capítulo se han descrito los conceptos de los diferentes medios existentes para alcanzar la confiabilidad en los sistemas. Asimismo, se ha presentado con detalle el estado actual de las técnicas propuestas para conseguir la tolerancia a fallos. Las técnicas han sido diferenciadas según su ámbito de operación: técnicas hardware, técnicas software, técnicas híbridas (hardware/software), y técnicas para FPGAs.

A pesar de que la mayoría de las propuestas basadas en hardware proveen una solución efectiva para la mitigación de fallos transitorios, por lo general estas técnicas conllevan una grave penalización al sistema en términos de recursos utilizados, consumo de potencia, área, tiempo de diseño y costes económicos. Por esta razón, su utilización no es viable en muchos casos, en especial, cuando se trata del diseño de sistemas embebidos que no cuentan con procesadores con grandes prestaciones.

Aunque las técnicas basadas en software redundante son más efectivas en cuanto al coste en comparación con las técnicas hardware, la redundancia software ocasiona dos importantes inconvenientes: el incremento de las necesidades de memoria y la degradación del rendimiento. La primera circunstancia puede limitar gravemente la aplicabilidad de la propuesta ya que se deben considerar las restricciones típicas de los sistemas embebidos y de los procesadores en los que están basados. El segundo inconveniente tiene una importancia relativa dependiendo del tipo de aplicación que se deseé endurecer. Sin embargo, merece la pena mencionar que algunas de estas técnicas software de tolerancia a fallos han sido utilizadas exitosamente para la mitigación de los efectos de la radiación en sistemas reales de misión crítica, como subsistemas de satélites y otros que participan en diferentes misiones espaciales.

En este contexto, surge la necesidad de diseñar técnicas híbridas (hardware/software) que sirvan para encontrar un punto medio óptimo entre las

técnicas puramente hardware y las puramente software. De hecho, las soluciones híbridas presentadas en este capítulo ofrecen resultados prometedores. No obstante, estas propuestas todavía siguen siendo *ad-hoc*, y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre las restricciones de diseño y los requisitos de confiabilidad.



Universitat d'Alacant
Universidad de Alicante

Capítulo 3

Co-endurecimiento: co-diseño hardware/software de estrategias de endurecimiento

En el capítulo anterior se ha evidenciado la creciente necesidad de la mitigación de los efectos de la radiación en los sistemas electrónicos modernos. De igual forma, se han estudiado las diferentes estrategias existentes en la actualidad para mitigar dichos efectos y se han identificado las principales ventajas e inconvenientes de cada una de ellas. Las técnicas basadas en hardware constituyen una solución efectiva para los fallos inducidos por radiación pero, al mismo tiempo, se traducen en incrementos en cuanto a recursos utilizados como: consumo de potencia, área, tiempo de diseño y costes económicos. Por otro lado, las técnicas basadas en software ofrecen una solución de bajo coste, pero también implican el incremento de las necesidades de memoria y la degradación del rendimiento de los sistemas. En cuanto a las estrategias híbridas, aunque ofrecen resultados prometedores con respecto a que permiten potenciar las ventajas y paliar los inconvenientes de cada uno de los dos escenarios (hardware y software), aún son muy específicas y poco flexibles.

Este capítulo propone aplicar los principios del co-diseño hardware/software para diseñar una estrategia híbrida de mitigación de los efectos de la radiación en los sistemas electrónicos. Esta estrategia está fundamentada en la combinación selectiva, incremental y flexible de enfoques de protección basados en hardware y software. De esta forma, será posible diseñar sistemas confiables a bajo coste, donde no sólo se satisfagan los requisitos de confiabilidad y las restricciones de diseño, sino que también se evite el uso excesivo de costosos mecanismos de protección.

El capítulo está organizado de la siguiente forma. En la Sección 3.1 se resumen los aspectos fundamentales del co-diseño hardware/software para el desarrollo de sistemas embebidos. A continuación, en la Sección 3.2 se propone la estrategia para el co-diseño de hardware/software de estrategias de endurecimiento, la cual hemos denominado *co-endurecimiento*. En la Sección 3.3 se define el procedimiento para realizar la exploración del espacio de diseño entre las técnicas de tolerancia a fallos basadas en software y las basadas en hardware. Por último, en la Sección 3.4 se recogen las principales conclusiones de este capítulo.

3.1. Co-diseño hardware/software

La complejidad de los sistemas embebidos es cada vez mayor, ya que el diseño de este tipo de sistemas involucra la consideración de cada vez más tipos de restricciones: rendimiento, tamaño, peso, consumo de potencia, confiabilidad, coste, tiempo de diseño, entre otros. Por este motivo, en las últimas décadas, los enfoques tradicionales de diseño han evolucionado hacia procesos más flexibles que permitan diseñar e integrar los complejos sistemas actuales.

Tradicionalmente el diseño de sistemas embebidos consta de dos procesos diferentes en los que se aborda el diseño del hardware y el software de forma completamente separada. El modo de trabajo más común consiste en un primer desarrollo de una especificación (muchas veces incompleta) que se describe en un lenguaje no formal y se traslada a los ingenieros de software y de hardware. Esto requiere un particionado del sistema a priori. Esto es, tomar la decisión de las tareas que se desean desarrollar usando software y las que se van a implementar mediante hardware. Superada esta fase del proceso, las decisiones de diseño, y en particular la partición hardware/software, deben mantenerse ya que cualquier cambio posterior implica el re-diseño del sistema. La Figura 3.1 presenta este proceso de diseño tradicional.

Los principales inconvenientes que aparecen asociados a este tipo de metodología de diseño pueden resumirse en:

- Carencia de representaciones unificadas tanto del hardware como del software, lo que complica el proceso de verificación del sistema en su totalidad y ocasiona la aparición de incompatibilidades a lo largo de la frontera entre el hardware y el software, dificultando la integración de ambas partes.
- Proceso previo de particionado hardware/software que, a pesar de que algunas veces está respaldado por la experiencia de los mejores profesionales, es susceptible de conducir a diseños diferentes a los que podrían considerarse como óptimos.
- Falta de un flujo de diseño bien definido, lo que dificulta los procedimientos de revisión de especificaciones, traduciéndose en inconvenientes durante el diseño e incrementos del tiempo de diseño.

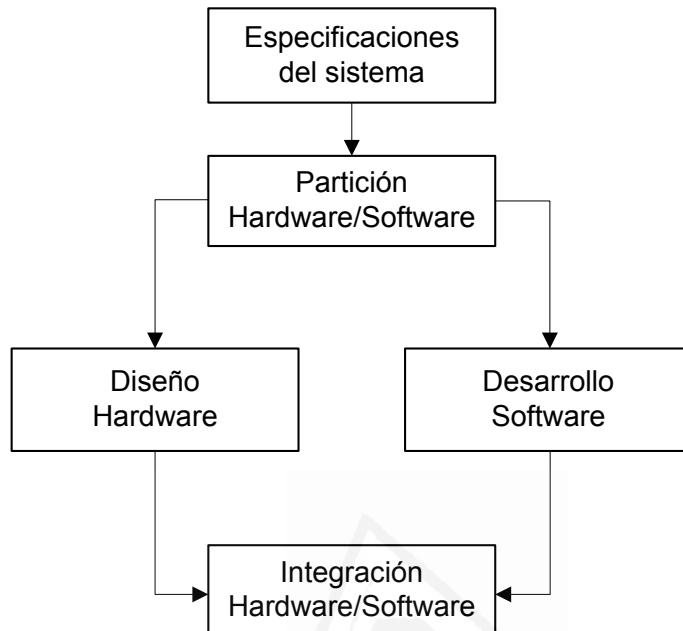


Figura 3.1: Proceso tradicional de diseño de sistemas embebidos

- Este proceso de diseño restringe la habilidad para explorar distintos niveles de compromiso entre el hardware y el software.

Para paliar estos inconvenientes ha surgido una nueva metodología de diseño: el co-diseño hardware/software. Según [De Micheli and Gupta, 1997], significa alcanzar los objetivos a nivel de sistema aprovechando la sinergia entre hardware y software a través de su diseño de forma concurrente e incremental. El objetivo principal del co-diseño hardware/software es el desarrollo de sistemas que no sólo cumplan con las especificaciones iniciales de funcionalidad, sino que además se optimicen las restricciones propias del diseño: rendimiento, coste, confiabilidad, entre otros [Wolf, 1994].

El hecho de utilizar una metodología que no optimice los factores mencionados puede llevar a un excesivo coste de diseño e implementación y, consecuentemente, a productos no competitivos en la sociedad actual. Es por ello que la aplicación de técnicas de co-diseño a los procesos de desarrollo supone una clara ventaja sobre las estrategias de diseño tradicionales.

A diferencia del enfoque tradicional, el co-diseño intenta mantener la interacción entre los procesos de diseño de hardware y desarrollo de software, evitando tener que hacer una partición temprana entre ambos. De esta forma, la fase de integración del sistema se va desarrollando de forma incremental e iterativa.

En la Figura 3.2 se presenta una adaptación del proceso de co-diseño hard-

ware/software propuesto en [Ernst, 1998]. En esta figura se aprecia el carácter incremental del co-diseño, al tener caminos de iteración una vez ha sido verificado el resultado obtenido. Asimismo, se puede ver que la exploración del espacio de diseño (hardware/software) se lleva a cabo en paralelo.

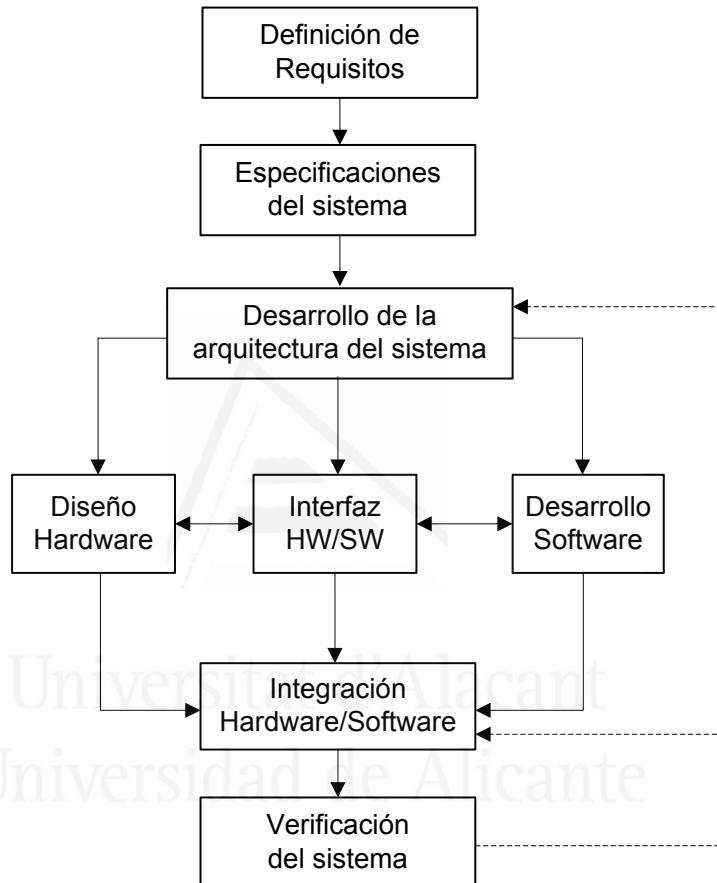


Figura 3.2: Proceso de co-diseño hardware/software de sistemas embebidos

En [De Micheli et al., 2002] se recogen los principales artículos publicados a nivel científico relacionados con el co-diseño hardware/software. Estos trabajos sientan las bases y principios de esta metodología desde sus primeros años.

A continuación se enuncian los principios más importantes que serán de utilidad para el desarrollo de nuestra propuesta:

- Exploración del espacio de diseño de forma flexible.
- Análisis de compromisos entre las diferentes restricciones y requisitos de diseño.

- Desarrollo hardware/software en paralelo.
- Proceso de integración hardware/software constante a medida que se desarrolla el sistema.
- Proceso de desarrollo iterativo e incremental.

En la siguiente sección se presenta nuestra propuesta para el diseño de técnicas híbridas de tolerancia a fallos utilizando el co-diseño.

3.2. *Co-endurecimiento hardware/software*

El proceso de co-diseño de un sistema hardware/software trata de encontrar un balance equilibrado entre la funcionalidad que desempeña cada uno de los componentes y los requisitos del sistema. De tal forma, que la partición hardware/software se realiza de forma iterativa buscando el compromiso óptimo entre los requisitos no funcionales. Por otro lado, el diseño de una estrategia de endurecimiento es un proceso complementario que busca la protección del sistema frente a los efectos de algún tipo de fallo.

En esta tesis doctoral se propone la aplicación combinada y selectiva de técnicas de protección hardware y software para co-diseñar una estrategia de endurecimiento de sistemas electrónicos frente a los fallos transitorios inducidos por radiación. Para ello se aplicarán los principios del co-diseño de sistemas al desarrollo de estrategias híbridas de mitigación de fallos, que denominaremos *co-endurecimiento* (*co-hardening* en inglés).

Aunque en algunos casos puede ser factible la combinación de técnicas hardware y software de mitigación de fallos para proteger los sistemas electrónicos, es común que esta estrategia dé como resultado un diseño sobreerdundante con características indeseables tales como: costes elevados, alto consumo de potencia y penalizaciones desproporcionadas en el rendimiento. Por lo tanto, en el *co-endurecimiento* se busca realizar una exploración del espacio de diseño de grano fino de las técnicas de protección hardware y software. Esto permite controlar selectivamente la aplicación de las técnicas de tolerancia a fallos y, de esta forma, insertar protecciones únicamente en las partes más vulnerables y críticas del sistema. Además, es posible seleccionar en qué parte del sistema será más efectiva la incorporación de mecanismos de redundancia, si en el hardware o en el software. Esta selección es muy importante ya que afecta al sistema de formas muy diferentes en términos de rendimiento, costes, confiabilidad, entre otros parámetros. De esta manera, será posible diseñar estrategias de protección a medida para cada sistema, que no sólo satisfagan los requisitos de confiabilidad y las restricciones de diseño, sino que también se evite el uso excesivo de costosos mecanismos de protección.

Resumiendo, a continuación se enuncian los principales principios del co-endurecimiento hardware/software para el diseño de estrategias híbridas de tolerancia a fallos:

- Basado en los principios del co-diseño de sistemas embebidos.
-

- Proceso dirigido por la aplicación y cuyo resultado es una solución a medida.
- Aplicación combinada de técnicas de endurecimiento hardware y software. Es decir, partición hardware/software de los mecanismos de protección del sistema.
- Endurecimiento selectivo en hardware y software, lo cual permite llevar a cabo una exploración de grano fino del espacio de diseño de las técnicas de tolerancia a fallos.
- Análisis de compromisos entre todos los requisitos del diseño y las exigencias de confiabilidad.

Para conseguir todo esto es necesario abordar dos problemas fundamentales. Por un lado, un flujo de co-diseño que permita la exploración eficiente del espacio de soluciones. Por otra parte, el desarrollo de herramientas que soporten este flujo (éstas últimas se presentarán en el siguiente capítulo).

3.3. Exploración del espacio de diseño de las técnicas de endurecimiento

Como en el co-diseño de sistemas, en el *co-endurecimiento* también se cuenta con múltiples alternativas tanto en la parte software como en el hardware, que es posible combinar para obtener diferentes compromisos entre los parámetros de diseño. Por esto, es necesario contar con un procedimiento definido para explorar de forma eficiente las distintas soluciones (exploración del espacio de diseño). En la Figura 3.3 se presenta el problema de la exploración del espacio de diseño de las técnicas de endurecimiento. Suponiendo que existen n técnicas de protección hardware que es posible implementar, se tendrán en total $n + 1$ posibles alternativas para el hardware del sistema, incluyendo la versión sin endurecer. Por otro lado, si en el software se tienen m técnicas de endurecimiento, serán $m + 1$ versiones del software que se deben evaluar. En total, $(n + 1) * (m + 1)$ posibles configuraciones hardware/software para implementar la estrategias de endurecimiento (incluyendo la versión del sistema hardware/software sin protección).

En este contexto, dependiendo del valor de n y m , podemos encontrarnos con un problema complejo en el que existen un gran número de posibilidades, y más aún si consideramos la posibilidad de aplicar varias opciones hardware o software de forma simultánea. La Figura 3.4 presenta el espacio de soluciones, donde cada uno de los puntos representa una posible implementación de la protección hardware/software del sistema.

Aplicando el flujo de diseño tradicional, la exploración se realizaría mediante el procedimiento descrito a continuación (Figura 3.5):

1. **Partición hardware/software:** como en el proceso de diseño tradicional, se realiza de forma temprana la partición hardware/software, pero en

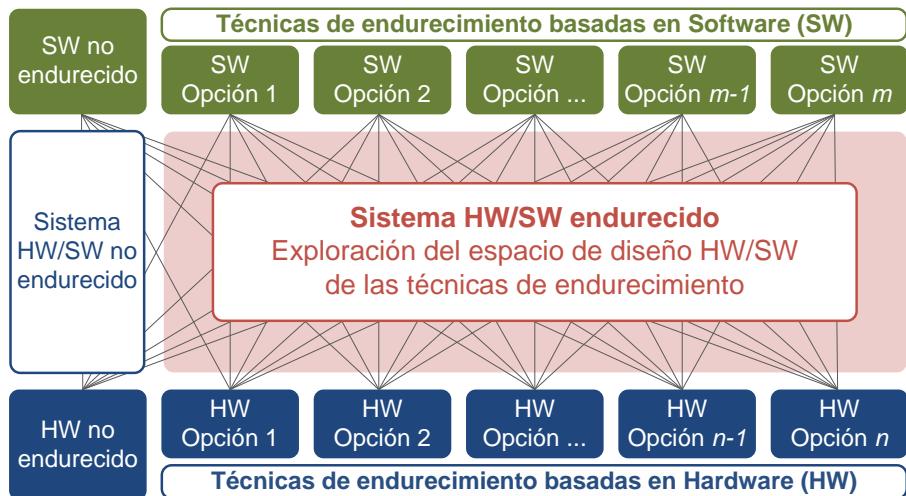


Figura 3.3: Exploración del espacio de diseño de las técnicas de endurecimiento

este caso, no se hace la partición de las funcionalidades del sistema sino de las tareas encargadas de la tolerancia a fallos.

2. **Diseño hardware y desarrollo software:** se realizan en paralelo el proceso de diseño de técnicas hardware de mitigación de fallos y el desarrollo de técnicas software, según haya quedado determinado en el paso anterior.
3. **Integración hardware/software:** se realiza la integración de los componentes hardware/software endurecidos del sistema. En este paso, podría llegar a descartarse alguna de las soluciones si, por ejemplo, su implementación supusiera superar el tiempo máximo de desarrollo requerido.
4. **Verificación funcional del sistema:** una vez se ha integrado el sistema, se procede a realizar su verificación funcional. Es decir, se debe garantizar que el endurecimiento hardware/software no ha causado que se modifique la funcionalidad original del sistema. En algunos casos, dependiendo de algunas incompatibilidades entre diferentes técnicas de protección, puede ocurrir que al combinarse, alteren el comportamiento esperado del sistema. Además, se debe reunir toda la información posible acerca de los parámetros de diseño y confiabilidad del sistema disponible hasta este momento. En este punto se pueden excluir del estudio diferentes configuraciones de técnicas hardware/software que resulten inviables para el endurecimiento del sistema, posiblemente porque modifican su funcionalidad o simplemente porque exceden el tope máximo de algún requisito de diseño (consumo de potencia, área, tamaño de memoria, coste, ...) o por el contrario, no alcanzan el umbral mínimo de alguno de los parámetros (rendimiento, frecuencia de operación, ...).

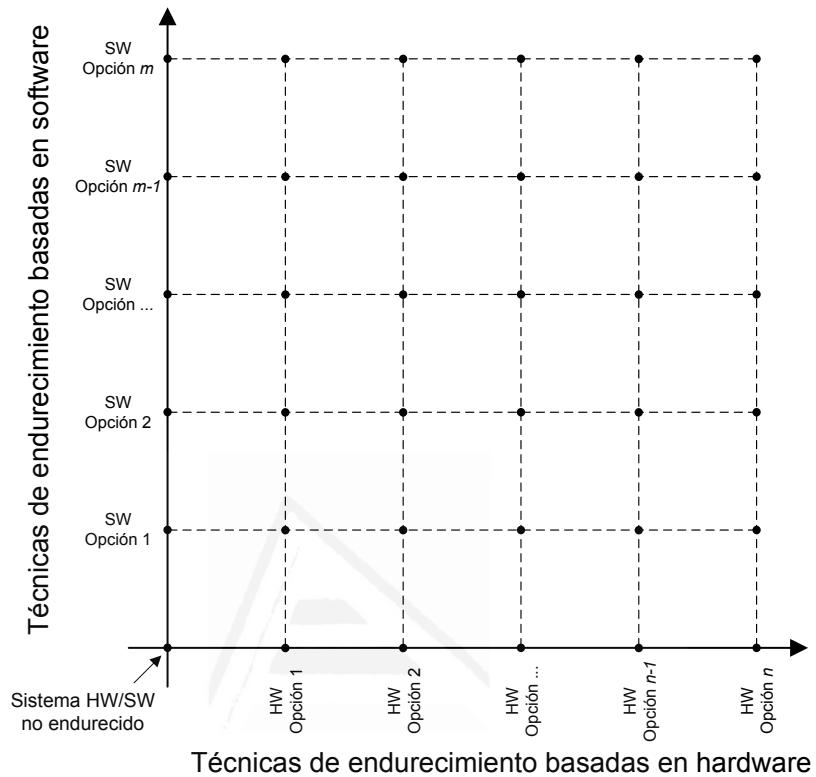


Figura 3.4: Espacio de diseño de las técnicas híbridas hardware/software de endurecimiento

5. **Evaluación de la confiabilidad:** las diferentes estrategias de endurecimiento diseñadas deben ser evaluadas para conocer los resultados de confiabilidad que ofrecen.
6. **Análisis de compromisos:** el procedimiento de exploración del espacio de diseño continúa con el análisis de compromisos entre todos los parámetros obtenidos hasta este punto. Estos parámetros pueden ser: consumo de potencia, área, tamaño de memoria, coste, rendimiento, frecuencia de operación, confiabilidad, fiabilidad, tasa de detección/recuperación de fallos, entre otros. Las decisiones tomadas durante estos análisis son motivadas principalmente por las especificaciones de la aplicación específica que se esté diseñando. Se busca encontrar el punto en el espacio de diseño de las técnicas de endurecimiento donde se satisfagan, de la mejor manera posible, todas las especificaciones del sistema, tanto las restricciones de diseño como los requisitos de confiabilidad.

Este tipo de soluciones, basadas en la combinación de técnicas hardware

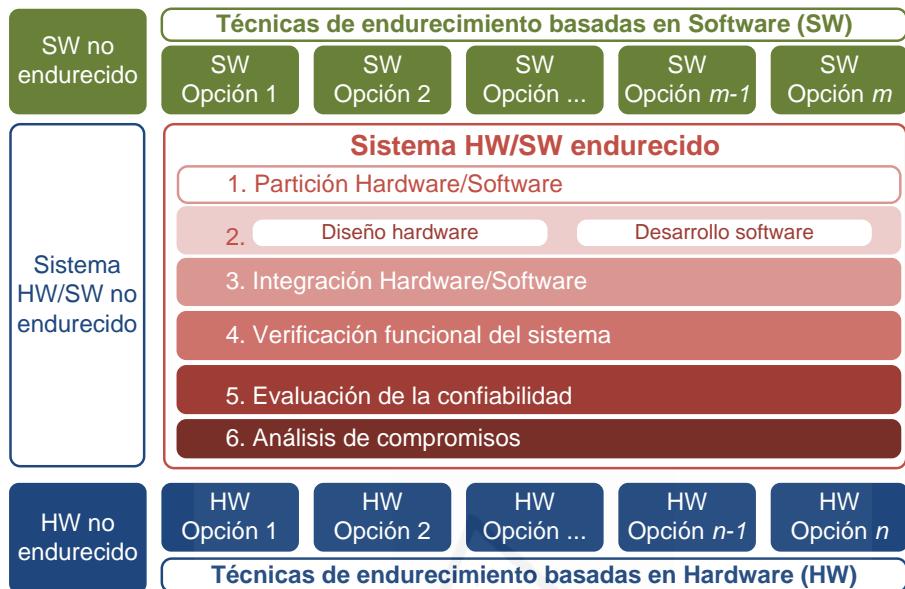


Figura 3.5: Proceso de exploración del espacio de diseño de las técnicas de endurecimiento

y software de tolerancia a fallos, pueden ser suficientes para el diseño de la estrategia de mitigación de fallos para muchas aplicaciones. Sin embargo, estas soluciones también suponen el uso excesivo de costosos mecanismos de redundancia, tanto en hardware como en software. Esto se traduce en inconvenientes como: degradación del rendimiento del sistema, necesidad de mayor capacidad de almacenamiento, crecimiento del área del circuito, incrementos en los tiempos de desarrollo y aumentos en los costes económicos.

Para evitar el uso excesivo de mecanismos de redundancia, y a su vez las consecuencias que esto conlleva, se propone explorar el espacio de diseño con un nivel de granularidad fino, es decir, a través de la implementación de las técnicas de endurecimiento de forma selectiva. En el software, el endurecimiento selectivo implica la protección parcial del programa centrando los esfuerzos en los procedimientos más críticos o en los recursos más susceptibles de fallos. En el hardware, esto puede significar proteger sólo algunos componentes del circuito, bien sea por su criticidad, o bien, con el ánimo de complementar la protección brindada por una estrategia basada en software.

En este contexto, por cada técnica de endurecimiento que previamente se contemplaba como un todo, se cuenta en realidad con varias técnicas selectivas diferentes basadas en la original, donde varía el nivel de protección que se implementa. Los mecanismos de redundancia se van incrementando gradualmente de una técnica selectiva a otra hasta alcanzar un nivel de máximo, que es equivalente a implementar la técnica de endurecimiento original de forma

plena.

De esta forma, el espacio de diseño de las técnicas de endurecimiento (presentado en la Figura 3.4) se ve ahora enriquecido al incluir todos los puntos posibles correspondientes a las nuevas posibles configuraciones hardware/software que ofrecen los enfoques de endurecimiento selectivo, como se puede apreciar en la Figura 3.6.

El hecho de incluir los enfoques selectivos en el espacio de diseño se traduce en una gran ventaja para el diseñador, ya que se ofrecen una gran cantidad de nuevas posibilidades y de puntos intermedios para explorar en el espacio de diseño. Lo que antes eran dos extremos, que consistían en tener la posibilidad de aplicar una técnica determinada o simplemente no aplicarla, ahora se convierte en un amplio rango de posibilidades intermedias, cada una de ellas con sus respectivas ventajas y desventajas. De esta forma, al explorar el espacio de diseño es posible realizar ajustes continuos a la estrategia de endurecimiento y afinar los parámetros del diseño hasta que se consiga el punto óptimo para cada aplicación específica.

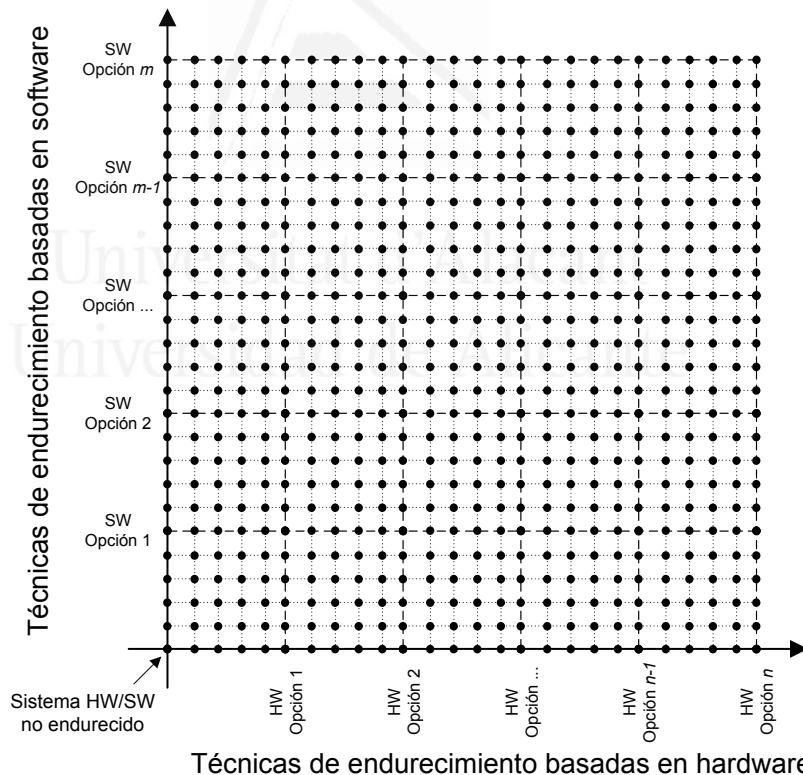


Figura 3.6: Espacio de diseño de las técnicas híbridas hardware/software de endurecimiento incluyendo enfoques selectivos

No obstante, al haber una mayor cantidad y concentración de posibilidades en el espacio de diseño, su exploración es más complicada y costosa. La exploración exhaustiva del espacio de soluciones resulta inviable en la mayoría de los casos. Por esta razón, en nuestra propuesta, el flujo de diseño del *co-endurecimiento* no se basa en una exploración exhaustiva del espacio de soluciones. Con el objetivo de disminuir considerablemente los tiempos de diseño, se propone un flujo de diseño guiado por la aplicación. A continuación se presenta con más detalle el flujo de diseño propuesto para el *co-endurecimiento*.

3.3.1. Exploración del espacio de diseño guiada por la aplicación

Teniendo en cuenta que la exploración exhaustiva del espacio de diseño de las técnicas de endurecimiento hardware y software (y aún más cuando se consideran los enfoques de protección parcial) resulta inviable en términos de tiempos de diseño, implementación y evaluación, se hace necesario reducir el espacio de exploración para llegar más rápidamente a un punto óptimo.

Por ello se propone un flujo de diseño basado en software para realizar el *co-endurecimiento*. Se trata de un flujo de co-diseño aplicable a cualquier tecnología de implementación (tanto ASIC como FPGA) y enfocado a la reducción de costes. En primer lugar, en términos de tiempo de desarrollo, ya que al reducir el espacio de soluciones se reduce también el tiempo necesario para la exploración. En segundo lugar, en términos de coste económico, ya que en el caso de una implementación final sobre FPGA, se podrían utilizar dispositivos *Flash* en lugar de las costosas FPGAs *rad-hard*. Para el caso de FPGAs basadas en memoria SRAM se requieren mecanismos adicionales para la protección de la memoria de configuración (ver Sección 2.5.4).

En la Figura 3.7 se observa el flujo propuesto para la exploración del espacio de *co-endurecimiento*.

De acuerdo a los principios del co-diseño de sistemas embebidos, el primer paso consiste en la completa *especificación del sistema*. Dentro de las especificaciones del sistema deben estar todas las *restricciones del diseño* y todos los *requisitos de confiabilidad* asociados a la aplicación específica que se esté diseñando. En general, las *restricciones de diseño* están relacionadas con parámetros como: área, peso, rendimiento, consumo de potencia, tamaño de memoria, y costes hardware. Los parámetros específicos concernientes a los *requisitos de confiabilidad* son aquellos relacionados con: fiabilidad, disponibilidad, seguridad, integridad, mantenibilidad, cobertura frente a fallos, tasa de detección de errores, tiempo de recuperación, entre otros. La correcta *especificación del sistema* es clave para el proceso de diseño de una estrategia de protección óptima: tanto las *restricciones de diseño* como los *requisitos de confiabilidad* motivarán y guiarán las decisiones tomadas durante el proceso de *co-endurecimiento*.

Una vez se haya especificado el sistema, se procede a seleccionar un conjunto de *esquemas de mitigación de fallos basados en software* que puedan resultar adecuados para el endurecimiento del sistema. Estas técnicas pueden ser aplicadas

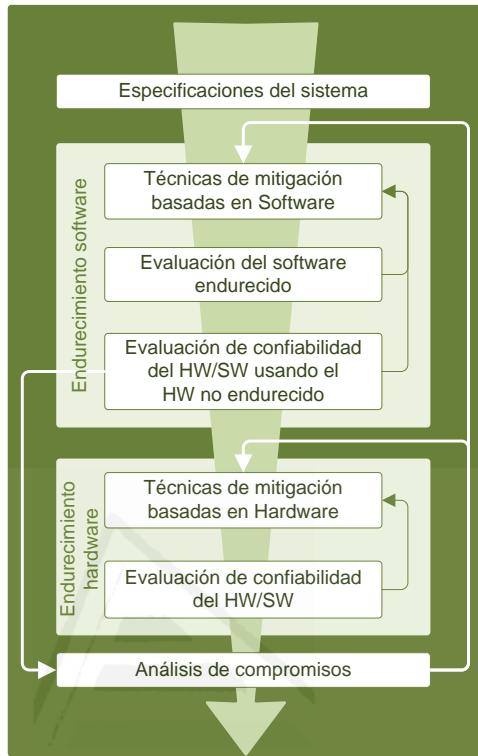


Figura 3.7: Flujo de diseño del *co-endurecimiento*

completamente a los programas o de forma selectiva, es decir, escogiendo únicamente algunas partes del software o de los recursos que éste utiliza para ser protegidos. Este proceso da como resultado un conjunto de implementaciones del software que son equivalentes funcionalmente, pero que se diferencian en el nivel de redundancia y la localización de las protecciones.

Seguidamente se realiza la *evaluación del software endurecido*. Esta evaluación tiene dos objetivos principales. El primero de ellos consiste en verificar que la aplicación de las técnicas de endurecimiento no ha modificado el funcionamiento de los programas. El segundo objetivo busca realizar una estimación preliminar de la confiabilidad de cada uno de los programas basada en simulaciones de fallos. De esta forma, para cada versión candidata del software: se verifica su funcionalidad; se obtiene el impacto del endurecimiento en términos de la degradación del rendimiento y el incremento del tamaño del código, en comparación con la versión del programa no endurecida (versión original del programa); y finalmente, se hace una valoración preliminar de la confiabilidad de los programas.

Basándose en esta evaluación preliminar, y de acuerdo a las especificaciones del sistema, se seleccionan las versiones software que sean más adecuadas

para la aplicación específica que se esté diseñando. Estas implementaciones son entonces sometidas al proceso de evaluación usando la implementación física real del sistema. Es decir, se hace la *evaluación de la confiabilidad del hardware/software usando el hardware no endurecido*. En este punto, el diseñador puede explorar diferentes niveles de compromiso entre parámetros del sistema como: tamaño del código fuente, rendimiento, y confiabilidad.

En el caso de que aún ninguna de las versiones software endurecidas satisfagan de forma óptima todas las restricciones del diseño o los requisitos de confiabilidad, la estrategia de endurecimiento debe ser complementada aplicando *técnicas de protección basadas en hardware*. Por lo tanto, el diseñador tiene que decidir cuál será la estrategia adecuada para proteger el hardware del sistema. Las técnicas hardware también pueden ser aplicadas a todo el diseño o de forma selectiva. Es posible que se busque únicamente la protección de las partes más vulnerables del diseño o insertar redundancia solamente para proteger aquellas partes del microprocesador donde es más complicado hacerlo mediante técnicas software. En este sentido, un nuevo parámetro debe ser tenido en cuenta entre los análisis de compromisos: el coste hardware (en términos de área, consumo de potencia y costes económicos).

Después de combinar los mejores candidatos de las implementaciones software con las diferentes versiones del hardware endurecido, se realiza la *evaluación de confiabilidad del hardware/software*, es decir, se evalúa la estrategia híbrida de protección.

Seguidamente, se continúa con el *análisis de compromisos* entre todos los parámetros disponibles en este punto del proceso de diseño. La solución no es necesariamente un punto único en el espacio de diseño, pueden existir un rango de configuraciones hardware/software que sean adecuadas para la aplicación. No obstante, en este punto el diseñador cuenta con información suficiente para determinar cual de todas es la mejor solución.

El proceso puede ser iterativo en caso de que el análisis de compromisos indique que la estrategia híbrida de mitigación de fallos debe continuar siendo afinada. Para esto se debe continuar con la exploración del espacio de diseño de las técnicas de protección, pero esta vez, realizando el proceso de diseño con un nivel de granularidad más fino, intentando ajustar las estrategias selectivas de protección tanto del software como del hardware.

Aunque tener definido el flujo de diseño para realizar adecuadamente la exploración del espacio de *co-endurecimiento* constituye uno de los factores claves para el diseño de estrategias híbridas de mitigación de fallos, resulta necesario, contar también con la asistencia de herramientas informáticas para soportar la propuesta y asistir al diseñador en las diferentes tareas relacionadas con la toma de decisiones y la evaluación de las estrategias. En el siguiente capítulo se identifican las herramientas necesarias para soportar esta propuesta.

3.4. Conclusiones

En la primera parte de este capítulo se han presentado los principios del co-diseño de sistemas, los cuales deben ser tenidos en cuenta debido a la creciente complejidad de los sistemas modernos. El auge de los sistemas embebidos en las últimas décadas ha provocado la evolución de los enfoques de diseño hasta el punto de madurez en el que se encuentran actualmente. En el co-diseño de sistemas se tienen en cuenta principios de diseño como: la exploración del espacio de diseño de forma flexible; el análisis de compromisos entre las diferentes restricciones y requisitos de diseño; el desarrollo hardware, software y su interfaz de forma iterativa e incremental; entre otros.

Fundamentado en lo anterior, se ha propuesto el *co-endurecimiento* hardware/software, el cual consiste en aplicar los principios del co-diseño de sistemas para diseñar técnicas híbridas hardware/software de tolerancia a fallos. El *co-endurecimiento* consiste en la combinación de enfoques de endurecimiento hardware y software para encontrar un punto medio donde se maximicen las ventajas y se minimicen las desventajas de cada uno de estos enfoques. Se propone además, la consideración de técnicas de endurecimiento parcial (selectivo) para realizar la exploración del espacio de diseño con un nivel de grano fino, donde sea posible encontrar un punto óptimo entre los todos los requisitos del diseño y los específicos de confiabilidad. Esta estrategia evita el uso innecesario y excesivo de costosos mecanismos de redundancia hardware y software. Sin embargo, para el *co-endurecimiento* se requiere solucionar dos problemas: una estrategia para reducir el espacio de diseño y explorarlo, ya que debido a la gran cantidad de posibilidades resulta inviable realizar una exploración exhaustiva; además, se requieren herramientas de asistencia al diseño y evaluación del *co-endurecimiento*.

Para solucionar el primer problema (relacionado con la estrategia adecuada para exploración del espacio de diseño de las técnicas de endurecimiento) se propone que el flujo de diseño sea guiado por las especificaciones de la aplicación que se esté diseñando y, además, se propone priorizar los enfoques de endurecimiento software teniendo en cuenta los elevados tiempos de diseño, implementación y evaluación de las técnicas hardware. De esta forma, la estrategia de endurecimiento, es complementada en una etapa posterior, mediante mecanismos hardware de redundancia. Este flujo de diseño agiliza el tiempo de diseño requerido para la estrategia de protección y, combinado con la aplicación selectiva de técnicas software y hardware, permite encontrar los puntos del espacio de diseño que satisfacen todas las especificaciones del diseño.

Para abordar el segundo problema, se propone conformar una infraestructura para el diseño, implementación y evaluación de técnicas híbridas de endurecimiento. Esta infraestructura está formada por: un entorno flexible para el desarrollo de técnicas software que será el encargado de guiar el proceso de *co-endurecimiento*; herramientas de terceros para el diseño de las técnicas hardware; y una herramienta para evaluar la confiabilidad ofrecida por las técnicas híbridas diseñadas. En el siguiente capítulo se presentan con detalle estas he-

rramientas.



Universitat d'Alacant
Universidad de Alicante



Universitat d'Alacant
Universidad de Alicante

Capítulo 4

Infraestructura para *co-endurecimiento*

El co-diseño hardware/software de técnicas de endurecimiento (*co-endurecimiento*) supone tener la flexibilidad para diseñar, implementar y evaluar estrategias de protección hardware, software, o híbridas. Teniendo en cuenta el flujo de diseño para la exploración del espacio de *co-endurecimiento* presentado en el capítulo anterior, este capítulo presenta los detalles de diseño, implementación y funcionamiento de las herramientas que conforman la infraestructura para el *co-endurecimiento* hardware/software de sistemas basados en microprocesador. La infraestructura integra utilidades propias y de terceras partes. En la Sección 4.1 se identifican las herramientas necesarias para soportar la propuesta presentada en el capítulo anterior. Seguidamente, la Sección 4.2 detalla el primer componente de la infraestructura: el entorno software desarrollado para el diseño y la aplicación automática de técnicas de tolerancia a fallos basadas en software. Esta herramienta es la encargada de guiar el proceso de *co-endurecimiento* mediante la exploración del espacio de diseño. A continuación, en la Sección 4.3 se presenta el segundo componente de la infraestructura: *FT-Unshades*, una herramienta incorporada para realizar la evaluación de la confiabilidad de sistemas hardware/software. Por último, en la Sección 4.4 se resumen las conclusiones más importantes del capítulo.

4.1. Herramientas requeridas para el *co-endurecimiento*

La creciente complejidad de los sistemas electrónicos modernos hace que cada vez resulte más necesaria la utilización de herramientas informáticas que asistan al diseñador para llevar a cabo las tareas del diseño y desarrollo de dichos sistemas. Tal es el caso también del desarrollo de técnicas de tolerancia a

fallos, donde los diseñadores suelen utilizar varios tipos de herramientas para diseñar y evaluar diferentes estrategias de endurecimiento.

Como se muestra en la Figura 4.1, al diseñar una estrategia híbrida hardware/software de mitigación de fallos, en general el diseñador debe contar como mínimo con herramientas que le permitan: desarrollar técnicas software de tolerancia a fallos, diseñar estrategias basadas en hardware para la mitigación de fallos, y por supuesto, herramientas que le permitan evaluar la confiabilidad ofrecida por las estrategias desarrolladas.

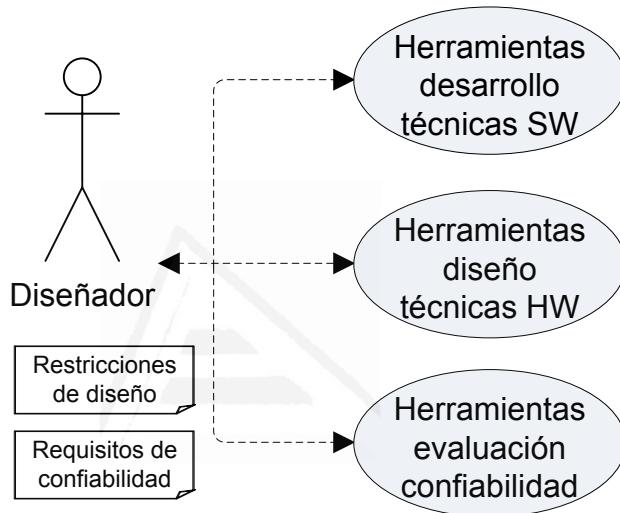


Figura 4.1: Infraestructura para el diseño y evaluación de técnicas híbridas de tolerancia a fallos

Con respecto a lo anterior, se ha identificado que actualmente existe una carencia de herramientas para el desarrollo de técnicas de endurecimiento basadas en software. Los investigadores que proponen este tipo de enfoques ofrecen soluciones puntuales que permiten aplicar únicamente sus propuestas. En algunas ocasiones realizan modificaciones sobre compiladores conocidos para aplicar transformaciones automáticas al código fuente de los programas [Oh et al., 2002c, Reis et al., 2005b], otras veces desarrollan herramientas *ad-hoc* desde cero para insertar la redundancia [Rebaudengo et al., 2001] o, incluso, en ocasiones aplican reglas de transformación del código fuente de forma manual [Rebaudengo et al., 2003]. Por esto, es necesario el desarrollo de una herramienta para el endurecimiento de software que permita el diseño, implementación y evaluación de diferentes técnicas de tolerancia a fallos basadas en software de manera general. Además deberá ser lo suficientemente flexible para permitir la aplicación selectiva y parcial de las técnicas de protección.

En cuanto a las herramientas para el diseño de técnicas de tolerancia a fallos basadas en hardware, como se ha mencionado en la sección 2.5.4, existen varias que permiten aplicar algunos mecanismos básicos de endurecimiento a

nivel del hardware. Estas herramientas se encuentran en un nivel de madurez apreciable, pudiéndose encontrar varias herramientas comerciales, como por ejemplo: *Mentor Precision Rad-Tolerant* [Mentor, 2010], *Synopsys Synplify Pro* [Synopsys, 2010], y *Xilinx TMR Tool (XTMR)* [Xilinx, 2009]. En este contexto, y teniendo en cuenta que el flujo de diseño del *co-endurecimiento* está dirigido por los enfoques basados en software y no aquellos basados en hardware, no se plantea el desarrollo de ninguna herramienta nueva para el diseño de técnicas hardware, sino que se puede hacer uso de herramientas de terceros que servirán para complementar el endurecimiento de los sistemas.

En lo que respecta a la evaluación de la confiabilidad, como se ha detallado en la Sección 2.6.1, existen una gran cantidad de propuestas y herramientas que permiten introducir fallos en el circuito de forma artificial, y observar su efecto, con el fin de realizar estimaciones de la confiabilidad del sistema mediante campañas de inyección de fallos. Con el fin de dar soporte a la propuesta de *co-endurecimiento*, ha incluido una herramienta de emulación de fallos para evaluar la confiabilidad de los sistemas hardware/software. En concreto, la herramienta que se utilizará para este propósito es *FT-Unshades* [Aguirre et al., 2005, Napoles et al., 2007]. Esta herramienta ha sido diseñada por el *Grupo de Ingeniería Electrónica de la Universidad de Sevilla* en España. Actualmente, el *Grupo UniCAD* (en cuyo seno se ha desarrollado esta tesis) de la *Universidad de Alicante* trabaja en estrecha colaboración este otro grupo de la ciudad de Sevilla.

En resumen, con el fin de conformar una infraestructura para el diseño y evaluación de técnicas híbridas de endurecimiento, se ha desarrollado un entorno flexible para el desarrollo de técnicas software que será el encargado de guiar el proceso de *co-endurecimiento*; se han utilizado herramientas de terceros para el diseño de las técnicas hardware; y se ha incluido una herramienta de evaluación de la confiabilidad diseñada por uno de los grupos de investigación con los cuales hemos venido colaborando en diferentes proyectos de investigación nacionales (*RENASER* y *RENASER+*). La Figura 4.2 muestra la infraestructura de trabajo propuesta.

4.2. Entorno para el endurecimiento de software: SHE

Con respecto al endurecimiento de software, la presente tesis doctoral se centra principalmente en la mitigación de fallos mediante técnicas basadas en redundancia a nivel de instrucción. A este respecto, a continuación se presentan dos cuadros comparativos con las principales características de las técnicas más relevantes que han sido presentadas en la Sección 2.5.2. Por un lado, la Tabla 4.1 reúne diferentes técnicas de detección de fallos, mientras que por otro lado, la Tabla 4.2 presenta técnicas de recuperación.

En relación con las tablas presentadas, se puede concluir lo siguiente con respecto a las técnicas software basadas en redundancia de instrucciones:

- Los métodos centrados en las instrucciones de bajo nivel (código ensam-

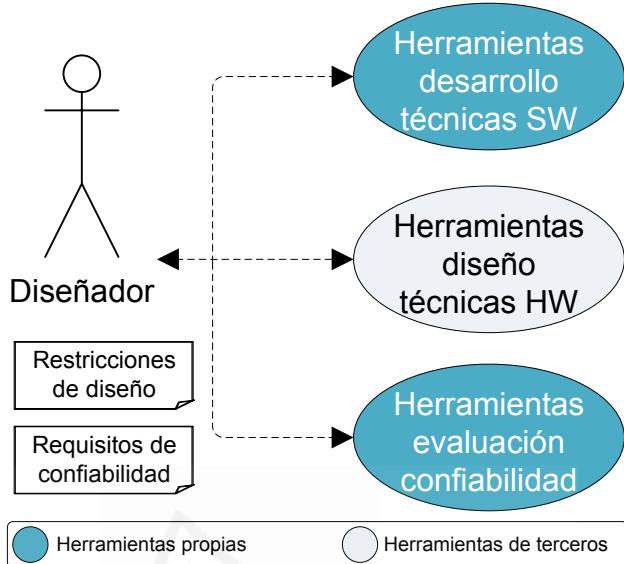


Figura 4.2: Herramientas (propias y de terceros) utilizadas para soportar la propuesta

Propuesta	Tipo Instrucciones	Arquitectura	Overhead Código	Overhead Datos	Overhead Tiempo	Fallos injectados detectados
ARBT	Alto nivel	Transputer T255 (RISC), 8051, MC68040, LEON	×5	×2	×3	63 %
EDDI	Bajo nivel	MIPS R10000 ISA-II, superescalar 4 vías	×1.5 – ×2	×2	< ×2	97 %
CFCSS	Bajo nivel	MIPS R4400 ISA-II, superescalar 4 vías	×1.2 – ×1.4	Sin Datos	×1.2 – ×1.7	96.9 %
SWIFT	Bajo nivel	VLIW	×2.4	Sin Datos	×1.4	100 % datos

Cuadro 4.1: Comparativo de técnicas software de detección de fallos

Propuesta	Tipo Instrucciones	Arquitectura	Overhead Código	Overhead Datos	Overhead Tiempo	Fallos injectados sin efecto negativo
ARBT-FT	Alto nivel	Microcontrolador Intel 8051	×2	×2 – ×3	×2.5	99.50 %
SWIFT-R	Bajo nivel	VLIW	Sin Datos	Sin Datos	×1.9	97.27 %

Cuadro 4.2: Comparativo de técnicas software de recuperación de fallos

blador) presentan un menor impacto sobre el tamaño del código y los datos de los programas, lo cual suele ser un factor crítico para el diseño de sistemas embebidos.

- El impacto en el tiempo de ejecución de los programas ocasionado por la aplicación de las técnicas de protección, también es menor para las propuestas basadas en la redundancia de instrucciones a bajo nivel.
- El impacto en el rendimiento puede mejorar al utilizar arquitecturas su-

perescalares y VLIW.

- Las técnicas de recuperación de fallos generan una importante contrapartida en relación al impacto que provocan en área y tiempo, por lo que las soluciones híbridas de redundancia (hardware y software) resultan interesantes, ya que podría encontrarse un compromiso óptimo entre área, rendimiento y confiabilidad.

Con el fin de contar con una herramienta que soporte la propuesta de diseño planteada, y teniendo en cuenta las conclusiones anteriores, se ha desarrollado una herramienta informática para el endurecimiento de software que permite el diseño, implementación y evaluación de diferentes técnicas de tolerancia a fallos basadas en software. El nombre que se le ha dado a esta herramienta es: SHE, del inglés *Software Hardening Environment*. A continuación se presentan sus características principales:

- Permite el diseño de técnicas software de tolerancia a fallos basadas en redundancia de instrucciones a bajo nivel, entendiendo por esto, el nivel de código ensamblador.
- Permite aplicar las técnicas diseñadas de forma automática en el código fuente de un programa dado. En otras palabras, recibe como entrada el código ensamblador de algún programa y genera automáticamente la versión endurecida de dicho código.
- Está basada en un núcleo genérico de endurecimiento que permite que el proceso de protección del software pueda llevarse a cabo de forma independiente a la arquitectura del microprocesador para el cual haya sido escrito el programa.
- El núcleo de endurecimiento expone al usuario una interfaz de programación dedicada al endurecimiento software (*Application Programming Interface (API)* de endurecimiento) la cual facilita la reutilización de código al proveer los métodos más comunes empleados en la programación de técnicas software de tolerancia a fallos, tales como: rutinas de análisis del grafo de control de flujo de ejecución de los programas y procedimientos para insertar código redundante en los programas.
- Es una herramienta flexible, ya que puede ser extendida con facilidad y se posibilita el diseño de técnicas software que pueden ser aplicadas de forma completa o selectiva a los diferentes recursos/componentes del programa.
- Admite que la salida (código ensamblador endurecido) pueda ser redirigida hacia otro microprocesador diferente a aquel para el cual había sido escrito el programa original. Esto es posible siempre y cuando se pueda hacer una equivalencia o adaptación entre todas las instrucciones que conforman el *Instruction Set Architecture (ISA)* de cada uno de los microprocesadores.

- También permite la evaluación preliminar de la confiabilidad proporcionada por las técnicas diseñadas.

Para contar con una herramienta que cumpla con las características anteriores, en primer lugar, se analizaron diferentes tipos de compiladores y herramientas existentes para verificar si alguna de estas herramientas podía ser adaptada al propósito del trabajo doctoral.

Los compiladores basados en una arquitectura concreta no resultaban adecuados para cumplir con el propósito de este trabajo, ya que eran muy específicos y altamente dependientes de la arquitectura. Por otro lado, se analizaron también los compiladores independientes de la arquitectura (compiladores *multi-target* tipo *GNU gcc* [FSF, 2011]), en los cuales hay una capa intermedia donde se realizan algunos procesos de forma genérica (independiente de la arquitectura), lo cual concuerda con el tipo de herramienta que se necesita para guiar el proceso de *co-endurecimiento*. Pese a que este tipo de compiladores están compuestos por un conjunto robusto de herramientas internas, tampoco resultaban adecuados para ser incorporados dentro de la infraestructura de endurecimiento, ya que estas herramientas, por lo general, están pensadas para implementar optimizaciones automáticas en tiempo de compilación y esto podría ocasionar problemas a la hora de implementar los mecanismos de redundancia. Además, en el caso de *GNU gcc*, carente de una API bien documentada y actualizada de su estructura interna, se dificultaba mucho la implementación de una nueva etapa intermedia en el proceso de compilación para implementar la redundancia.

Por lo tanto, se decidió desarrollar una herramienta informática que cumpliera con todas las características mencionadas partiendo desde cero (*Software Hardening Environment - SHE*), ya que al hacerlo de esta forma se tiene el control total y el conocimiento completo de qué ocurre en cada una de las etapas de la compilación, y de esta forma, es posible satisfacer todos los requisitos y funcionalidades mencionadas antes. No obstante, no se descarta en el futuro la posible integración de *Software Hardening Environment* (SHE) en compiladores de alto nivel como *GNU gcc* o *LLVM/Clang*, ya que éstos compiladores pueden generar código ensamblador, lo cual constituye el punto de entrada a nuestro entorno de endurecimiento software.

SHE ha sido diseñado usando ANTLR 3.2 [Parr, 2011] (un entorno para el desarrollo de intérpretes, compiladores y traductores, a partir de la descripción de gramáticas). El lenguaje de programación utilizado para realizar la implementación de la herramienta es C#.

SHE constituye una herramienta para el diseño, implementación y evaluación de técnicas software de tolerancia a fallos basadas en redundancia de instrucciones a bajo nivel. El entorno propuesto está compuesto por tres capas principales de herramientas o módulos software: un conjunto de *front-ends* de compilación, un *núcleo de endurecimiento genérico*, y un conjunto de *back-ends* de compilación. La Figura 4.3 muestra el esquema general del entorno de endurecimiento software.

Dado un cierto programa en ensamblador para una de las arquitecturas so-

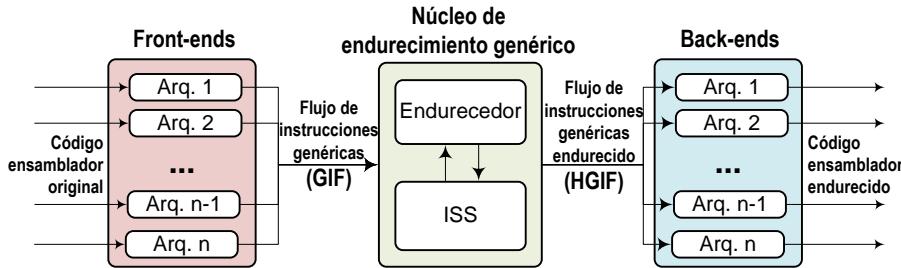


Figura 4.3: Entorno para el endurecimiento software (*Software Hardening Environment* - SHE)

portadas por SHE (*Arq. 1*, *Arq. 2*, ..., *Arq. n*), cada *front-end* del compilador asociado a una arquitectura concreta de microprocesador toma el código ensamblador nativo, realiza el análisis léxico, sintáctico y semántico del código y genera un *Flujo Genérico de Instrucciones* conocido como *Generic Instruction Flow* (GIF). Este flujo representa una abstracción de alto nivel del programa que es soportada por la arquitectura genérica de microprocesador sobre la cual opera el *núcleo de endurecimiento genérico*. A continuación, se realizan las tareas de endurecimiento sobre el GIF con las herramientas disponibles en la capa del *núcleo de endurecimiento genérico* (el *endurecedor* y el simulador del conjunto de instrucciones o *Instruction Set Simulator* (ISS)), y se genera un nuevo GIF endurecido o *Hardened Generic Instruction Flow* (HGIF). Por último, este nuevo flujo se traslada mediante uno de los *back-ends* a código ensamblador nativo de cualquiera de los microprocesadores soportados. Estos *back-ends* realizan transformaciones directas entre el conjunto de instrucciones de la arquitectura genérica y el repertorio de instrucciones del microprocesador seleccionado.

Nótese la flexibilidad del entorno de endurecimiento, ya que usando este esquema, es posible procesar un código ensamblador escrito para un microprocesador determinado, y obtener código endurecido para la misma arquitectura o bien, re-dirigir la salida para otro microprocesador diferente.

Las principales ventajas de SHE se mencionan a continuación y serán ampliadas con detalle en las siguientes secciones:

- Posee un núcleo de endurecimiento basado en una *arquitectura genérica de microprocesadores*, lo cual permite al entorno de endurecimiento y al usuario del API, trabajar con múltiples microprocesadores diferentes de forma transparente.
- El API del *núcleo de endurecimiento genérico* permite diseñar, implementar, aplicar y evaluar diferentes técnicas software de tolerancia a fallos de forma independiente de la plataforma.
- Posee la flexibilidad necesaria para facilitar el *endurecimiento selectivo del software*, es decir, es posible aplicar el endurecimiento sólo a los recur-

sos más críticos del microprocesador, dependiendo de la aplicación que ejecuta.

4.2.1. Arquitectura genérica de microprocesadores

Una vez realizadas las comprobaciones léxicas, sintácticas y semánticas de cualquier programa escrito para una arquitectura específica cualquiera, se realiza el proceso de transformación del dominio de una arquitectura específica al dominio de la arquitectura genérica propuesta. Dichas tareas son responsabilidad de cada uno de los *front-ends* del compilador. Algunos compiladores conocidos funcionan también de esta manera, como por ejemplo: *GNU gcc* o *LLVM/Clang*.

La *arquitectura genérica de microprocesadores* provee el espacio de trabajo para el *núcleo de endurecimiento genérico*. Reúne todos los elementos comunes de diferentes arquitecturas con el fin de facilitar el diseño e implementación de las técnicas de tolerancia a fallos actuales basadas en software con independencia del microprocesador.

Las características principales de la arquitectura genérica son:

- Está definida por medio de instrucciones genéricas.
- Posee herramientas para gestionar los recursos de la arquitectura mientras se realizan modificaciones al código original en tiempo de compilación para añadir la redundancia.
- Permite la identificación y análisis del grafo de control de flujo de los programas.

Instrucciones genéricas

La arquitectura genérica está definida por medio de instrucciones genéricas y recursos genéricos (registros, *flags*, memoria principal). Una instrucción genérica es una instrucción que forma parte del ISA de la arquitectura genérica de microprocesador.

Estas instrucciones se construyen a partir de las instrucciones originales del ISA nativo, elevando su nivel de abstracción para lograr añadir una mayor cantidad de información. En este sentido, las instrucciones genéricas pueden ser consideradas como un *envoltorio* de las instrucciones originales que contiene información enriquecida que permite su interacción con la arquitectura genérica. Este *envoltorio*, constituye una capa de abstracción del hardware que permite trabajar con diferentes microprocesadores objetivo dentro del *núcleo de endurecimiento genérico*. En otras palabras, en la arquitectura genérica no se hace distinción alguna entre un conjunto dado de instrucciones genéricas que han sido construidas a partir de ISAs de microprocesadores diferentes. Esto hace posible aplicar las técnicas de endurecimiento a los programas de manera independiente del microprocesador para el cual hayan sido escritos (siempre y

cuando se cuente con un *front-end* y *back-end* específico para dicho microprocesador).

En la Figura 4.4 se presentan los campos que conforman una instrucción genérica y a continuación se explica con mayor detalle cada uno de ellos.



Figura 4.4: Campos que conforman una instrucción genérica

- **Label.** Es la etiqueta asignada a la línea de código ensamblador en la que se encuentra la instrucción. Este campo aumenta la legibilidad del código producido a la hora de trazar las transformaciones de código realizadas por SHE. Puede no estar asignado (campo vacío).
- **Address.** Contiene la información necesaria para localizar una instrucción genérica en un espacio de direccionamiento determinado.
- **OpCode.** Es el código de la palabra reservada de la instrucción del lenguaje original (ensamblador). Se utiliza únicamente a nivel informativo, y para ser posteriormente utilizado por el *back-end*. No obstante, es un elemento muy valioso para trazar los resultados que produce SHE.
- **Instruction Type.** Es uno de los campos más importante desde el punto de vista del endurecimiento, ya que es el encargado de clasificar las instrucciones de acuerdo a su función. Durante el endurecimiento, las trasformaciones software a efectuar dependen del tipo de instrucción. Por ejemplo, las instrucciones aritméticas, las de almacenamiento, o las que impliquen saltos pueden verse afectadas por transformaciones distintas. Los tipos de instrucción definidos en la arquitectura genérica se presentan en la Figura 4.5. Aunque en la presente tesis doctoral se trabaja únicamente con arquitecturas escalares (y de ahí los tipos de instrucción presentados), también están previstas otro tipo de arquitecturas, como las de los procesadores vectoriales.

Los primeros tres tipos de instrucciones (*Flow Control*, *Interrupt* y *Directive*) son comunes a todas las arquitecturas. Los siguientes poseen un nombre compuesto por la palabra SCALAR, que indica que se refiere a arquitecturas escalares, y a continuación un campo que indica el tipo de instrucción (por ejemplo, aritméticas, lógicas, desplazamientos, almacenamiento, entrada/salida). A pesar de que el *instruction type* también da soporte a otro tipo de arquitecturas, como las vectoriales, en esta memoria se presenta los tipos asociados a arquitecturas escalares, objeto de estudio en este trabajo doctoral.

- **Generic Operand List.** Es una lista enlazada de operandos genéricos, presentes en la instrucción original. Esta lista se puede observar en la Figura

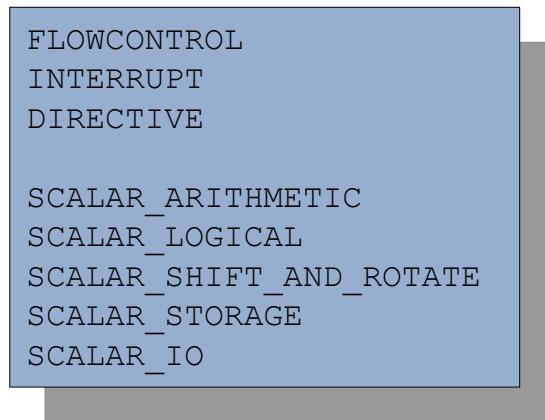


Figura 4.5: Tipos de instrucciones genéricas para procesadores escalares

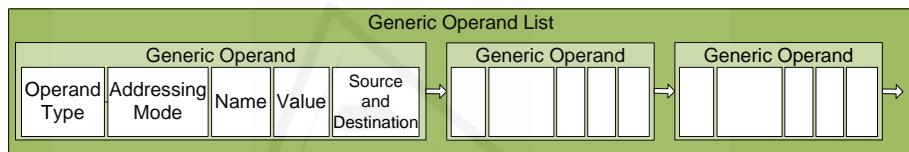


Figura 4.6: Lista de operandos genéricos

4.6, donde cada operando genérico se encuentra definido por los siguientes campos.

- **Operand Type.** Define el tipo de operandos genéricos con los que se va a trabajar. Estos pueden ser: *Register*, *Literal*, *Address*, *Constant* y *Flag*.
- **Addressing Mode.** Establece el modo de direccionamiento del operando genérico. Puede ser: *Absolute*, *PC Relative*, *Register Indirect*, *Secuencial*, *Conditional*, *Skip*, *Register*, *Base Plus Offset and Variations*, *Immediate Literal*, *Implicit*, *Absolute Direct*, *Indexed Absolute*, *Base Plus Index*, *Base Plus Index Plus Offset*, *Scaled* y *None*.
- **Name.** Es la etiqueta (nombre) original del operando en la arquitectura específica.
- **Value.** Si el operando representa un valor numérico, este valor será almacenado en este campo.
- **Source and Destination.** Es una bandera que sirve para indicar si el operando actúa dentro de la instrucción como fuente y destino de la operación al mismo tiempo. Por ejemplo, en una instrucción como ADD r1, 01; el operando r1 se utiliza como fuente y destino al mismo tiempo, ya que el resultado de esta instrucción es equivalente a: r1 = r1 + 01.

- **Affected Generic Flag List.** Corresponde a la lista enlazada de los *flags* genéricos que serán afectados tras la ejecución de la instrucción. Esta lista se define como se presenta en la Figura 4.7.

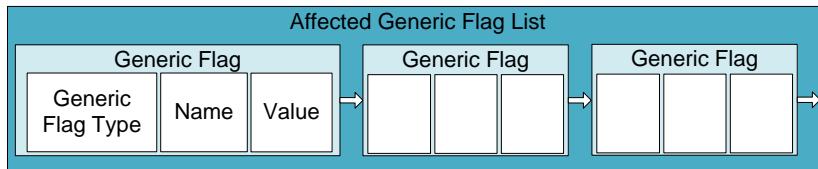


Figura 4.7: Lista de *flags* genéricos afectados por la instrucción

- **Generic Flag Type.** Define el tipo de *flag* genérico con el que se va a trabajar. Los diferentes tipos de *flag* son: *Zero*, *Not Zero*, *Carry*, *Not Carry*, *Signus*, *Not Signus*, *Overflow*, *Not Overflow*, *Interrupt Enable*, *Not Interrupt Enable*, *CPU ON*, *CPU OFF*, *OSC ON*, *SCG*, *Other* y *No ne*.
- **Name.** Es el nombre del *flag* en la arquitectura específica.
- **Value.** Es el valor booleano del *flag* correspondiente. Es útil para trazar la ejecución de las instrucciones.
- **Comment.** Corresponde al comentario original de la línea de código ensamblador. Se preserva para que el *back-end* restablezca la documentación o los comentarios del código fuente original.

Cabe anotar que las clasificaciones establecidas, para tipificar las instrucciones, los modos de direccionamiento, los tipos de *flags* y los tipos de operandos, se encuentran abiertos para ser completados en el momento que alguna arquitectura específica así lo requiera.

Gestión de recursos

La segunda característica más importante de la arquitectura genérica corresponde a las facilidades que ofrece para la gestión de los recursos de la arquitectura. Dentro de la arquitectura genérica se pueden representar recursos físicos propios de microprocesadores concretos. Recursos como: el banco de registros, la memoria de programa, diferentes tipos de memoria de datos (*cache*, *scratchpad*, *RAM*, etc.), *flags* del microprocesador, entre otros. De estos es importante subrayar las posibilidades que se ofrecen para realizar la gestión de la memoria de programa y del banco de registros.

Gestión de la memoria de programa. El API proporcionado por la arquitectura genérica permite la identificación del mapa de memoria del programa ensamblado, el cual puede estar compuesto por varias secciones de memoria (Figura 4.8). Asimismo, se ofrece la posibilidad de mantener actualizado el mapa

de memoria mientras se realizan modificaciones al código ensamblador durante la compilación. Esto es necesario ya que al realizar operaciones de endurecimiento de *software* es muy probable tener que insertar nuevas instrucciones dentro del código original afectando el mapa de memoria original.

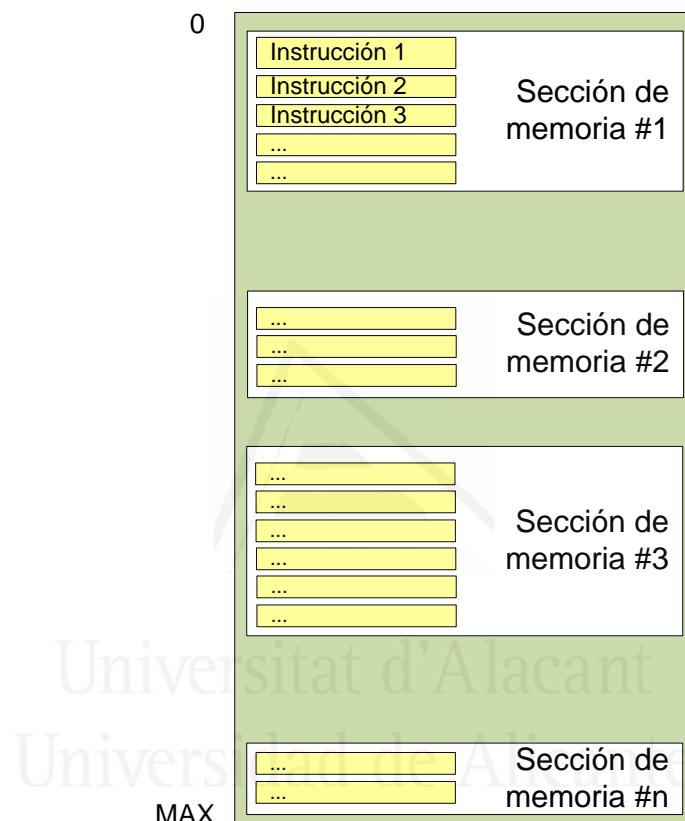


Figura 4.8: Mapa de memoria de programa

Una vez el programa ha sido ensamblado, se identifica su mapa de memoria. Si en este punto se hace necesario realizar la inserción de nuevas instrucciones, la gestión de la memoria de programa se hace de forma consistente. Sin embargo, esta tarea implica realizar constantemente actualizaciones a las instrucciones que habían sido ensambladas previamente en el mapa de memoria, por lo que resulta ser una tarea compleja de realizar. Para esto, se proporcionan las funciones para poder realizar la gestión de memoria teniendo en cuenta estas dificultades, soportando la realización de cambios de forma dinámica sobre el mapa de memoria. Se definen tres tipos diferentes de movimientos que se llevan a cabo en las secciones de memoria al insertar una nueva instrucción: dilatación, desplazamiento y redistribución. A continuación se explica cada uno

de ellos. Estas operaciones deben mantener la coherencia de ejecución del programa, es decir, deben mantener funcionalmente invariante la ejecución del programa.

- **Dilatación.** Al insertar una o más instrucciones nuevas en alguna de las secciones de memoria, se debe producir una *dilatación* de dicha sección de memoria. Es decir, dicha sección de memoria debe crecer en la medida en que se hayan insertado instrucciones, asignando las direcciones de ensamblado a las nuevas instrucciones y actualizando las direcciones de las instrucciones originales. En la Figura 4.9 se puede observar una representación gráfica de este proceso.

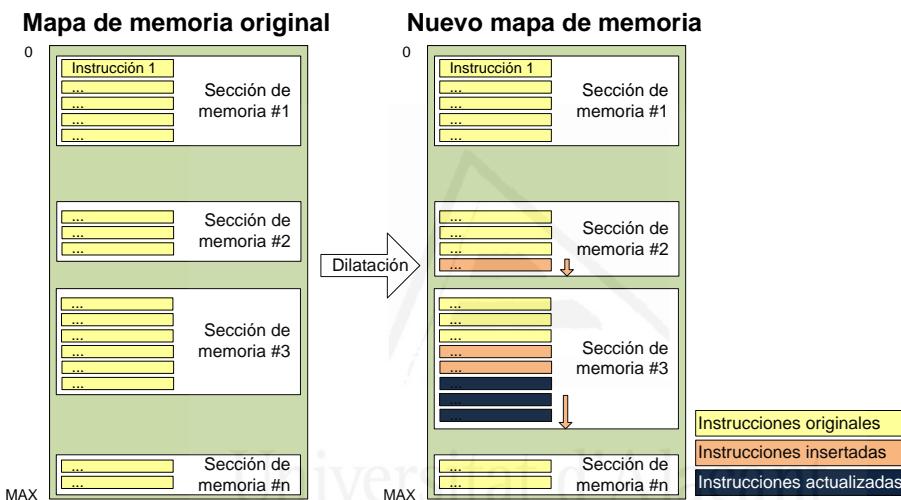


Figura 4.9: Dilatación de secciones de memoria

- **Desplazamiento.** Un *desplazamiento* de una sección de memoria se produce cuando al insertar nuevas instrucciones en una determinada sección, ésta se dilata hasta el punto en que se superpone a la siguiente sección de memoria y se hace necesario *desplazar* completamente esta última sección y actualizar todas las direcciones. La Figura 4.10 presenta un ejemplo gráfico de este caso.
- **Redistribución.** Como resultado de una o varias dilataciones y/o desplazamientos en las secciones de memoria ocasionados por la inserción de nuevas instrucciones, puede llegar a ocurrir que se desborde la memoria, causando que la última sección de memoria crezca por encima de los límites físicos de la memoria. Por lo tanto, se hace necesario realizar una *redistribución* de todas las secciones de memoria, actualizando todas las direcciones de las instrucciones, ubicando cada sección de memoria justo después de la anterior, es decir, eliminando los espacios libres existentes entre secciones de memoria ("huecos") en el mapa de memoria. Este

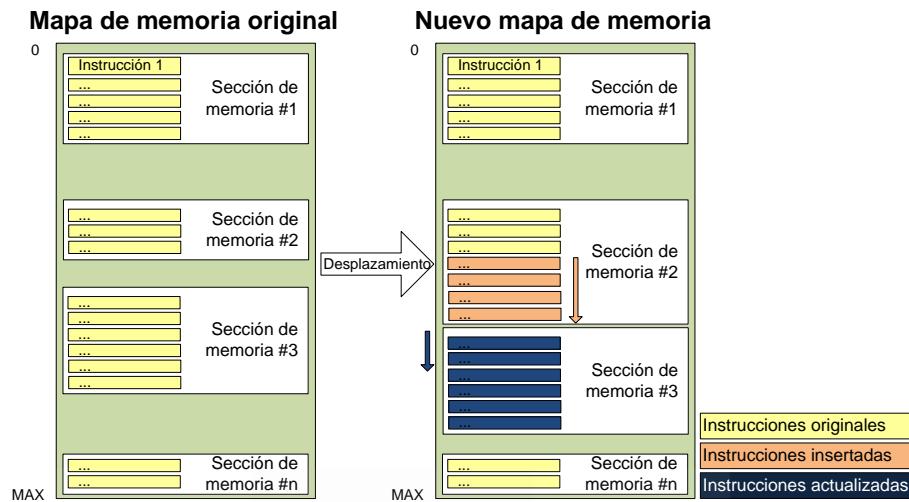


Figura 4.10: Desplazamiento de secciones de memoria

Este proceso es necesario para intentar encajar todas las instrucciones ensambladas dentro de los límites físicos de la memoria. La Figura 4.11 presenta un ejemplo de *redistribución*.

Este último, en términos de rendimiento es el peor de los casos ya que todas las instrucciones deben ser reubicadas para optimizar los recursos

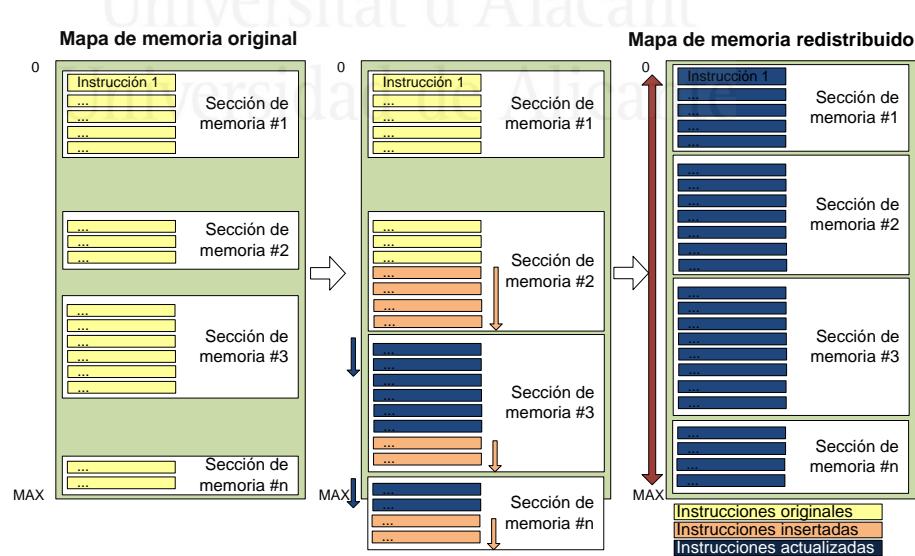


Figura 4.11: Redistribución de secciones de memoria

físicos de la memoria.

Estas tareas son posibles ya que en este trabajo, así como en otras propuestas, se asume que el código que se va a endurecer no aprovecha la reserva de memoria dinámica, es decir, cada estructura de datos está definida de forma estática en tiempo de compilación. Esto no constituye una limitación importante para los desarrolladores de sistemas embebidos, quienes muchas veces se ven obligados a evitar precisamente este tipo de prácticas por estándares de codificación industrial como [MISRA, 2004].

Gestión del banco de registros. Considerando que muchas de las técnicas software de endurecimiento se basan en la duplicación/triplicación de los datos, la arquitectura genérica ofrece también la posibilidad de gestionar un banco de registros genérico, donde se puede encontrar toda la información de los registros internos del microprocesador y su utilización. Como se observa en la Figura 4.12, el banco de registros está formado por una lista enlazada de registros genéricos.

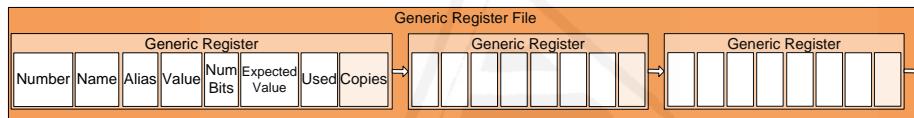


Figura 4.12: Banco de registros genérico

Cada registro genérico contiene la siguiente información:

- **Number.** Es un número que identifica de forma única al registro dentro de la arquitectura genérica.
- **Name.** Es el nombre que recibe el registro en la arquitectura original.
- **Alias.** Si el registro ha sido renombrado en el programa, o puede ser referenciado con más de un nombre dentro de la arquitectura original o el código del programa, *alias* almacena dicho nombre.
- **Value.** Contiene el valor numérico que almacena el registro (si éste es conocido).
- **Num Bits.** Indica el ancho en bits del registro.
- **Expected Value.** Este campo se utiliza para verificar el proceso del endurecimiento en simulación, ya que permite comparar el valor de un registro con su valor esperado (si éste se conoce). De esta forma, se puede determinar si la aplicación de una técnica determinada de endurecimiento ha modificado la funcionalidad del programa o no.
- **Used.** Es una bandera que indica si el registro está siendo utilizado en el programa.

- **Copies.** Contiene a su vez, otra lista enlazada de registros genéricos que almacena los registros que son copias del actual. Este campo es útil para poder controlar de forma consistente el proceso de replicación de registros.

Grafo de control de flujo

Otra de las funciones clave para muchas técnicas de endurecimiento software está basada en la posibilidad de identificar y realizar análisis del *Grafo de Control de Flujo* de ejecución de los programas (*Control Flow Graph - CFG*) [Oh et al., 2002b, Oh et al., 2002c, Reis et al., 2005b]. El análisis del control de flujo determina las estructuras de control de un programa y de esta forma permite construir el grafo de control del flujo [Yau and Chen, 1980]. La arquitectura genérica diseñada también ofrece estas posibilidades.

El CFG es un grafo dirigido en el que los nodos son *bloques básicos* y las aristas representan los cambios del flujo de control de la ejecución del programa. Para construir un CFG, en primer lugar, se deben identificar los *bloques básicos* que posee el programa. Un *bloque básico* es un conjunto de instrucciones que se ejecutan de forma consecutiva (sin saltos) en las que el flujo de control entra al principio y sale al final. Los *bloques básicos* se determinan por:

- Sólo la última instrucción puede ser una instrucción de salto o llamada a función.
- Sólo la primera instrucción puede ser el destino de un salto o llamada.

En un CFG no se tiene información de los datos. Por tanto, una arista en el grafo significa que el programa puede tomar dicho camino, o en otras palabras, significa que el flujo de control de la ejecución del programa puede pasar por ese camino en un momento determinado. En la Figura 4.13 se puede apreciar un ejemplo del CFG de un programa de 14 instrucciones I_i que conforman 5 nodos (*bloques básicos*).

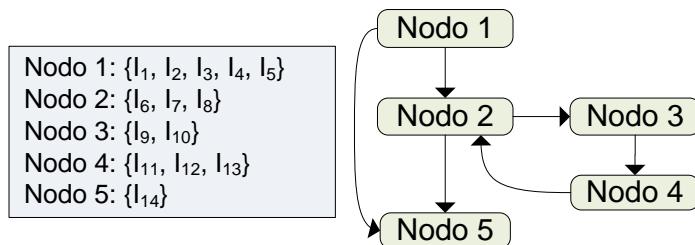


Figura 4.13: Ejemplo de un grafo de control de flujo (CFG)

4.2.2. Núcleo de endurecimiento genérico

El *núcleo de endurecimiento genérico* está construido en torno a la *arquitectura genérica de microprocesador* y contiene dos componentes principales: el *endure-*

cedor y el simulador del repertorio de instrucciones (en adelante ISS). A continuación se presenta cada uno de ellos.

Endurecedor

La responsabilidad del *endurecedor* es doble. Por un lado, integra las rutinas usuales de mitigación de errores (que pueden ser extendidas por el usuario) e incluso permite aplicar dichas rutinas de forma selectiva (por ejemplo: aplicarlas selectivamente a diferentes conjuntos de recursos del microprocesador). Esto es posible ya que dispone de una API interna de programación de técnicas de endurecimiento software, la cual está compuesta por los métodos que usualmente se utilizan para desarrollar técnicas de endurecimiento software. Esta API permite al diseñador, el desarrollo de nuevos métodos de tolerancia a fallos basados en software, o incluso permite experimentar con nuevas versiones mejoradas de las técnicas existentes. Los métodos implementados en la API tienen un nivel de complejidad variable, encontrándose subrutinas básicas, como por ejemplo, la inserción de una instrucción genérica dentro de una posición específica del flujo de instrucciones genéricas - GIF; hasta procedimientos más complejos como la inclusión en el código de procedimientos de recuperación; o incluso existen métodos de la API que implementan técnicas de tolerancia a fallos completas que pueden ser configuradas a partir de los parámetros que se le entreguen a las funciones. En la Figura 4.14 se puede observar un ejemplo de programación que usa la API propuesta. En este caso, se presenta una versión simplificada del método “*Harden*”, que sirve para seleccionar una de las técnicas de endurecimiento implementadas para ser aplicada a los programas. Este método posee los siguientes parámetros: *oriArch*, que es una instancia de la clase *MicroGenArch*, la cual implementa la arquitectura genérica de microprocesador presentada antes; *hArch*, una instancia de la misma clase, que será utilizada como salida del método y será la que almacene el programa endurecido; *method* es una cadena de caracteres que sirve para seleccionar la técnica de tolerancia a fallos a aplicar en el programa; también recibe otros parámetros para configurar diferentes detalles del endurecimiento.

Por otro lado, el *endurecedor* es el responsable de aplicar a los programas las técnicas diseñadas de forma automática, teniendo en cuenta las opciones de endurecimiento seleccionadas por el usuario. Una vez que alguno de los *front-ends* de compilación ha generado el GIF, el *endurecedor* permite incorporar características de tolerancia a fallos a los programas según las técnicas de detección y recuperación de fallos implementadas y disponibles en la infraestructura. Es importante recordar que la generación del código endurecido es automática y sigue reglas de transformación a nivel de instrucción. Por último, el resultado final del proceso ejecutado por el *endurecedor* genera un nuevo flujo de instrucciones genéricas, pero en este caso, dicho flujo ha sido endurecido (HGIF) según la técnica seleccionada y sus parámetros de configuración.

A continuación se presentan las opciones más importantes disponibles en el *endurecedor* para que el diseñador pueda seleccionar y parametrizar de forma adecuada la técnica de endurecimiento que desea aplicar a su programa:

```

public static bool Harden(MicroGenArch oriArch,
                         out MicroGenArch hArch, string method, ...)
{
    hArch = (MicroGenArch)oriArch.Clone()
    hArch.program.GenInsFlow = new GenInsFlow()
    oriArch.FindMemoryMap()

    switch(method) {
        case "TMR1":
            TMR1(oriArch, hArch, ...)
        case "TMR2":
            TMR2(oriArch, hArch, ...)
        case "SWIFT-R":
            SWIFT_R(oriArch, hArch, ...)
            ... // otras técnicas de endurecimiento
    }
    return true
}

```

Figura 4.14: Ejemplo de programación con la API de endurecimiento: método *Harden*

1. **Método (-m, --method [=VALUE])**. Permite seleccionar la técnica de endurecimiento que se desea aplicar.
2. **Procedimiento de recuperación (--voter [=VALUE])**. Selecciona el procedimiento para realizar votación por mayoría, y en caso que sea posible, realizar la recuperación de un fallo.
3. **Microprocesador (--mcpu [=VALUE])**. Sirve para identificar el microprocesador en el cual funcionará el programa endurecido, es decir, se selecciona el *back-end* del compilador apropiado.
4. **Salida (--output [=VALUE])**. Nombre del archivo de salida endurecido después de llevar a cabo el proceso de endurecimiento.
5. **Nivel de replicación de registros (--replicationRegisterLevel [=VALUE])**. *VALUE* es un número que indica el nivel de redundancia para los registros, de este modo:
 - 0 - Sólo redonda registros absolutamente necesarios (por defecto). Por ejemplo en la instrucción ADD s0, s1 sólo s0 será redundado.
 - 1 - Redunda todos los registros involucrados en la instrucción. Por ejemplo en la instrucción ADD s0, s1 serán redundados s0 y s1.

6. **Nivel de replicación de instrucciones** (`--replicationTimes [=VA-LUE]`). Indica el número de veces que debe redundar las instrucciones.

- 0 - No hace ninguna copia (sin redundancia).
- 1 - Una copia por instrucción (duplica).
- 2 - Dos copias por instrucción (triplica).

La utilidad de las dos opciones anteriores radica en las posibilidades que ofrecen para parametrizar y personalizar alguna técnica de endurecimiento pre-diseñada.

Simulador del repertorio de instrucciones - ISS

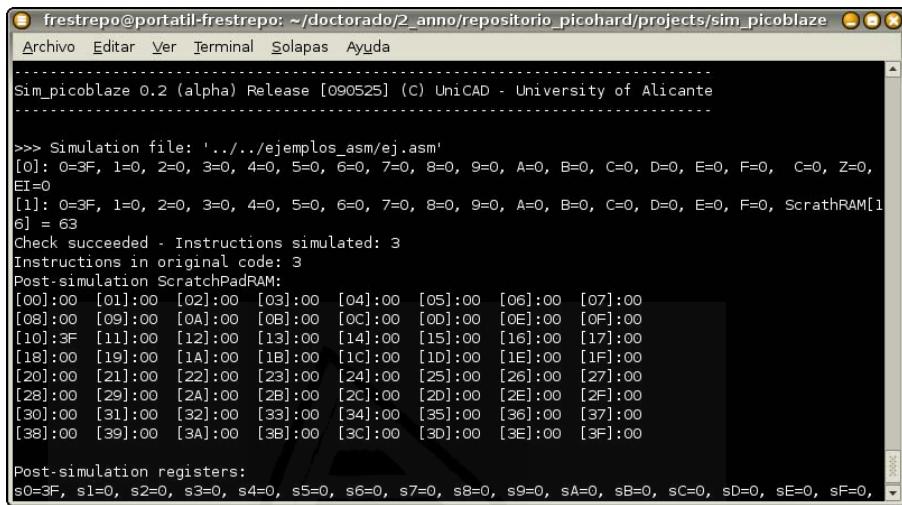
El segundo componente del *núcleo de endurecimiento genérico* es el simulador del repertorio de instrucciones (*Instruction Set Simulator - ISS*). Una vez compilados los programas por alguno de los *front-ends* de compilación, el ISS recibe el flujo de instrucciones genéricas y realiza las tareas de simulación sobre éste, por tanto, se trata de un simulador de instrucciones genéricas.

El ISS asiste al diseñador en las tareas comunes de depuración de los programas simulados. No obstante, sus funcionalidades más relevantes tienen que ver con la asistencia que se le brinda al diseñador para desarrollar y evaluar técnicas de endurecimiento software, y además, la información que ofrece, permite explorar el espacio de diseño del lado software con mayor facilidad. Las principales funcionalidades del ISS se resumen en los siguientes tres puntos:

1. Como es usual para los simuladores de repertorios de instrucciones, el ISS presenta la información acerca del estado de los recursos físicos del microprocesador (modelados mediante la *arquitectura genérica de microprocesadores*) durante y después del proceso de simulación de los programas. De esta forma, se pueden depurar con facilidad los programas al conocer el estado de los recursos (registros del banco de registros, memorias, pila de ejecución, contador de programa, *flags*, etc.) durante la simulación de cada una de las instrucciones del programa, así como su estado final una vez terminada la simulación.

La Figura 4.15 presenta la captura de pantalla de la simulación de un programa sencillo, listando en la ejecución paso a paso, el estado de los registros (en hexadecimal) y los *flags* después de la simulación de cada instrucción. Por ejemplo, al simular la primera instrucción del programa se puede observar lo siguiente: [0] : 0=3F, 1=0, 2=0, 3=0, 4=0, 5=0, 6=0, 7=0, 8=0, 9=0, A=0, B=0, C=0, D=0, E=0, F=0, C=0, Z=0, EI=0, donde: [0] es el valor del contador del programa, a continuación se presenta la lista de pares registro - valor en hexadecimal, enumerados de 0 hasta F, por último se presenta la lista de pares flag - valor (C = Carry; Z = Zero; EI = Enable Interrupt). En caso de que se haya modificado alguna posición de memoria por la instrucción simulada, se

presentan el estado de los registros y el estado de la posición de memoria que ha cambiado. Por último, después de la ejecución paso a paso, una vez concluida la simulación, se presenta el estado final de todas las posiciones de la memoria y los registros.



```
frestrepo@portatil-frestrepo: ~/doctorado/2_anoo/repositorio_picohard/projects/sim_picoblaze
Archivo Editar Ver Terminal Solapas Ayuda
sim_picoblaze 0.2 (alpha) Release [090525] (C) UniCAD - University of Alicante
>>> Simulation file: '../../ejemplos_asm/ej.asm'
[0]: 0=3F, 1=0, 2=0, 3=0, 4=0, 5=0, 6=0, 7=0, 8=0, 9=0, A=0, B=0, C=0, D=0, E=0, F=0, C=0, Z=0,
EI=0
[1]: 0=3F, 1=0, 2=0, 3=0, 4=0, 5=0, 6=0, 7=0, 8=0, 9=0, A=0, B=0, C=0, D=0, E=0, F=0, ScrathRAM[1]
E] = 63
Check succeeded - Instructions simulated: 3
Instructions in original code: 3
Post-simulation ScratchPadRAM:
[00]:00 [01]:00 [02]:00 [03]:00 [04]:00 [05]:00 [06]:00 [07]:00
[08]:00 [09]:00 [0A]:00 [0B]:00 [0C]:00 [0D]:00 [0E]:00 [0F]:00
[10]:3F [11]:00 [12]:00 [13]:00 [14]:00 [15]:00 [16]:00 [17]:00
[18]:00 [19]:00 [1A]:00 [1B]:00 [1C]:00 [1D]:00 [1E]:00 [1F]:00
[20]:00 [21]:00 [22]:00 [23]:00 [24]:00 [25]:00 [26]:00 [27]:00
[28]:00 [29]:00 [2A]:00 [2B]:00 [2C]:00 [2D]:00 [2E]:00 [2F]:00
[30]:00 [31]:00 [32]:00 [33]:00 [34]:00 [35]:00 [36]:00 [37]:00
[38]:00 [39]:00 [3A]:00 [3B]:00 [3C]:00 [3D]:00 [3E]:00 [3F]:00
Post-simulation registers:
s0=3F, s1=0, s2=0, s3=0, s4=0, s5=0, s6=0, s7=0, s8=0, s9=0, sA=0, sB=0, sC=0, sD=0, sE=0, sF=0,
```

Figura 4.15: Captura de pantalla del ISS

2. Además de la información usual que ofrecen los simuladores tradicionales, el ISS presenta información complementaria que facilita el proceso de exploración del espacio de diseño de las técnicas de endurecimiento software. A continuación se exponen las características más importantes que ofrece el ISS para este fin:

- Después del proceso de simulación de un programa, se presenta un informe con una caracterización de las instrucciones que han sido simuladas durante la ejecución del programa. La caracterización consiste en contabilizar las instrucciones que han sido simuladas y clasificarlas de acuerdo al tipo de instrucción (aritméticas, lógicas, desplazamientos, rotaciones, control de flujo, etc.). Esta información es útil para decidir el tipo de protección que debe implementarse en un programa, ya que la estrategia de protección puede variar dependiendo del tipo de programa, tal y como quedará demostrado en el siguiente capítulo. Por ejemplo, si se desea endurecer un programa que haga un uso intensivo de operaciones aritméticas, puede ser adecuada la aplicación de una técnica de endurecimiento determinada, mientras que si se trata de un programa con muchas instrucciones de control de flujo, la técnica adecuada a implementar puede ser distinta.

- Para ayudar al diseñador a identificar los recursos más críticos que utiliza el programa, el ISS ofrece también un informe de utilización de recursos una vez haya terminado la simulación del programa. Finalmente, se presenta un listado con el número de veces que se ha utilizado cada uno de los registros del banco de registros. En este informe se presenta el número de veces que cada uno de los registros ha sido objeto de operaciones de: lectura, escritura, o lectura/escritura. Esta información es útil para prever el incremento, en términos de tiempo de ejecución y tamaño de código, que se produce cuando es protegido cada uno de estos registros. Añadir redundancia a un registro que sea accedido constantemente puede ocasionar incrementos que no sean adecuados para cumplir con los requisitos de la aplicación.
Además, a partir de esta información se presenta también el *tiempo de vida (lifetime)* de cada uno de los registros. El *tiempo de vida* representa el tiempo en el que un registro almacena información útil durante la duración del programa. Cualquier fallo que ocurra en el registro durante este tiempo destruye la integridad del dato almacenado [Lee and Shrivastava, 2009b]. Por lo tanto, el *tiempo de vida* sirve como una estimación de la vulnerabilidad de cada registro, y el conjunto de todos los *tiempos de vida* es útil para estimar la vulnerabilidad de todo el banco de registros. De esta forma, el diseñador puede priorizar cuales son los registros que deben ser protegidos, en caso de que no sea posible añadir redundancia a todo el banco de registros.
- Si los resultados del estado final de los recursos del microprocesador son conocidos de antemano por el diseñador, es posible incorporarlos al código fuente del programa mediante un *pragma* del compilador. Esto permite verificar, de forma automática, si el programa ha obtenido los resultados esperados una vez haya terminado su simulación. Esta opción tiene gran utilidad para depurar los programas y programar *scripts* de verificación de su funcionalidad.
- De forma complementaria a la opción anterior, y con el fin de verificar el proceso de endurecimiento de los programas, el ISS ofrece la posibilidad de verificar que los resultados obtenidos por las versiones endurecidas de los programas coinciden con los resultados originales. Esto permite cotejar que la funcionalidad de los programas endurecidos sigue siendo la misma que la de su versión no-endurecida y que no se ha modificado funcionalmente al aplicarle el endurecimiento. Es posible comprobar el estado de los recursos que elija el diseñador, tales como: los resultados finales almacenados en los registros del banco de registros y las memorias de datos, o el histórico de los datos que hayan sido escritos en los puertos de salida durante la simulación del programa.
- Cuando se verifica la funcionalidad de la versión endurecida de un

programa con su versión no-endurecida, el ISS presenta también un informe con los incrementos, en términos de tamaño de código y tiempo de ejecución, sufridos por el programa al ser endurecido. Aunque el incremento del tamaño del código es muy importante debido a las limitaciones usuales de memoria de programa de algunos microprocesadores usados en aplicaciones de los sistemas embebidos, desde el punto de vista de la tolerancia a fallos es más crítica la información referente al incremento del tiempo de ejecución del programa, ya que el tiempo de ejecución adicional será tiempo en el cual el microprocesador será susceptible a que ocurra algún posible fallo (pese a que se haya aplicado algún tipo de endurecimiento, seguirán existiendo partes de la microarquitectura del microprocesador que son vulnerables y que no pueden ser protegidas mediante técnicas software).

3. Por otra parte, el ISS permite evaluar de forma preliminar la confiabilidad proporcionada por las técnicas de endurecimiento software. Para este fin, el ISS ofrece la posibilidad de simular fallos tipo SEU durante la simulación del programa. Cada fallo se simula mediante un *bit-flip* (cambio del estado lógico de un bit) en los recursos de la arquitectura. Para evaluar la confiabilidad de un programa se debe diseñar una campaña de simulación de fallos, donde se hagan múltiples simulaciones del programa, simulando un único *bit-flip* en cada una de ellas, observando los resultados obtenidos y comparándolos con los resultados esperados.

El ISS tiene acceso a los recursos más importantes de la arquitectura, tales como: banco de registros, contador de programa, puntero de pila de ejecución, *flags*; y por ende, puede simular fallos en cualquiera de estos recursos. Sin embargo, no tiene acceso a los registros internos de la microarquitectura, como aquellos ubicados en el *pipeline* del microprocesador. Por lo tanto, los resultados de confiabilidad obtenidos mediante el ISS deben ser considerados como estimaciones preliminares (pero muy realistas, según se demostrará en el Capítulo 5), los cuales sirven para poner a punto las técnicas de endurecimiento software y hacer comparaciones entre los resultados de confiabilidad que brinda cada una de ellas. No obstante, para obtener resultados más precisos acerca de la confiabilidad de los sistemas, se debe incluir una herramienta donde, por un lado, se tenga acceso completo a todos los recursos del microprocesador (incluidos los registros de la microarquitectura), y por otro lado, se puedan evaluar también los resultados obtenidos al endurecer el hardware del sistema (el ISS sólo admite evaluar la confiabilidad del software).

Con esto, el conjunto *Endurecedor+ISS* ofrece un rico conjunto de opciones y parámetros de co-diseño que pueden ser usados para realizar una exhaustiva exploración del espacio de diseño del lado del software, como se detallará en el capítulo 5.

4.2.3. Endurecimiento selectivo del software

Otra de las ventajas de SHE es que permite que las técnicas de endurecimiento software puedan ser aplicadas de forma selectiva, es decir, aplicando redundancia sólo a las partes más vulnerables de los programas. Este hecho amplia las posibilidades existentes para proteger los programas, ya que muchas técnicas que previamente podían ser aplicadas únicamente en su totalidad, ahora pueden ser adaptadas para ser aplicadas de forma selectiva a los programas. Por lo tanto, se ofrece mayor flexibilidad al diseñador al ofrecerle un espacio de diseño con más posibilidades, donde se pueden encontrar puntos intermedios de protección en los que se satisfagan de forma óptima los requisitos de la aplicación que se esté diseñando. Por ejemplo, si al aplicar una técnica de endurecimiento concreta en un programa, se excede el requisito que establece su máximo tiempo de ejecución; entonces se puede aplicar esta misma técnica pero de forma selectiva, esta vez protegiendo únicamente los recursos o secciones más críticas; y de esta forma, se puede conseguir que el programa endurecido cumpla con la restricción de tiempo establecida sin degradar demasiado la cobertura frente a fallos.

Para lograr implementar esta característica del endurecimiento selectivo del software, se ha utilizado el concepto de *esfera de replicación* o *Sphere of Replication* (SoR), el cual ha sido implementado de forma flexible. En el siguiente apartado se presentan detalles al respecto.

Esfera de replicación - SoR

El concepto de *esfera de replicación* o *Sphere of Replication* (SoR) fue propuesto por [Reinhardt and Mukherjee, 2000]. Esta esfera define el dominio de ejecución redundante, es decir, los elementos que se ubiquen dentro de la SoR se consideran que poseen mecanismos de redundancia y que están protegidos frente a fallos. Por lo tanto, la SoR define la frontera de protección de las técnicas de endurecimiento.

La implementación de la SoR en SHE es flexible, porque permite modificar el nivel de protección de las diferentes técnicas de tolerancia frente a fallos, al incluir o excluir diferentes componentes de la esfera de ejecución redundante. Por ejemplo, se puede incluir/excluir el subsistema de memoria, el banco de registros, o incluso seleccionar sólo un subconjunto de registros críticos del banco de registros.

El *endurecedor* clasifica de modo especial las instrucciones cuya ejecución implica que haya un flujo de datos a través de la frontera de la SoR. Cuando una instrucción provoca la entrada de un dato en la SoR (por ejemplo: mediante la lectura de un puerto, la carga de literal en registro, o lectura de un dato almacenado en memoria), dicha instrucción es etiquetada como *inSoR*; consecuentemente, si se produce un flujo de datos hacia afuera de la esfera, es decir, que hay información abandonando la esfera de protección (por ejemplo: la escritura en un puerto, o la escritura de un dato en una memoria externa), se etiqueta como *outSoR* (ver Figura 4.16). Esta clasificación de las instrucciones

facilita la implementación de las estrategias adecuadas de protección para cada uno de estos casos.



Figura 4.16: Esfera de replicación - SoR

Las fronteras de la *SoR*, y en consecuencia, la cobertura de la protección, pueden cambiar según la técnica aplicada. Por ejemplo, en el caso de la técnica EDDI [Oh et al., 2002c], se considera que todo el subsistema de memoria se encuentra dentro de la *SoR*, y por lo tanto, las instrucciones responsables de ejecutar operaciones de lectura/escritura en la memoria no ocasionan que haya ningún flujo de información hacia dentro/fuera de la esfera. En este sentido, si el subsistema de memoria estuviera considerado externo a la *SoR*, como en el caso de SWIFT [Reis et al., 2005b], las mismas instrucciones encargadas de las operaciones de lectura/escritura de la memoria sí producirían un flujo de datos que atravesie (en ambos sentidos) la frontera de la esfera, y por tanto, deberían ser manipuladas de forma especial por la técnica de endurecimiento.

Al flexibilizar el concepto de la *SoR*, como se propone en esta tesis doctoral, se ha hecho posible realizar el endurecimiento software de forma selectiva. Por ejemplo, es posible tener parcialmente protegido el banco de registros (algunos registros del banco se consideran dentro de la *SoR* y otros por fuera de ella). En este caso particular, la identificación de aquellas instrucciones que ocasionan un flujo de datos hacia dentro/fuera de la esfera no es tan fácil, ya que los flujos de datos a través de la esfera pueden ocurrir en cualquier instrucción. Por ejemplo, en la instrucción `ADD s0, s1`; donde el registro `s0` se considera que está ubicado dentro de la esfera, y el registro `s1` se considera por fuera de la esfera; al leer el dato almacenado en `s1` para sumarlo al dato almacenado en `s0`, se está produciendo un flujo de datos desde afuera de la esfera hacia adentro de ella, y por lo tanto, la instrucción `ADD` debe clasificarse como *inSoR*. De la misma manera, en el caso de que `s0` estuviera considerado como externo a la esfera, y `s1` dentro de la esfera, el flujo de datos que se produciría al leer `s1` y sumarlo con el dato almacenado en `s0`, sería hacia afuera de la esfera, y por consiguiente, la instrucción `ADD` debería ser clasificada como *outSoR*.

Partiendo de la base de que únicamente los fallos pueden propagarse y convertirse en errores cuando los datos corruptos abandonan la *SoR*, se recomienda que las técnicas de endurecimiento software analicen con especial cuidado las instrucciones que generan un flujo de datos hacia fuera de la *SoR* (las instrucciones *outSoR*), y que se verifique la integridad de los datos previamente a que hayan abandonado la esfera de protección. Para facilitar esto, SHE subdivide los nodos (bloques básicos) del grafo de control de flujo (CFG) en subnodos

justo después de cada instrucción clasificada como *outSoR*. De esta forma, se pone a disposición del diseñador de las técnicas de endurecimiento, un CFG enriquecido con información adicional que sugiere los puntos donde se deberían realizar las comprobaciones para garantizar que los datos son válidos cuando abandonan la esfera. La Figura 4.17 presenta gráficamente la subdivisión de un nodo en dos subnodos diferentes, debido a que el nodo original contiene una instrucción de tipo *outSoR*.

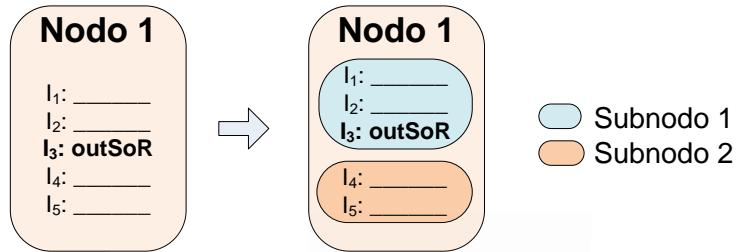


Figura 4.17: Subdivisión de los nodos del CFG en subnodos

4.3. Herramienta para la evaluación de la confiabilidad: *FT-Unshades*

El segundo componente principal de la infraestructura para el *co-endurecimiento* es la herramienta para la evaluación de la confiabilidad llamada *FT-Unshades* [Aguirre et al., 2005, Napoles et al., 2007]. Esta herramienta ha sido diseñada por el *Grupo de Ingeniería Electrónica* de la *Universidad de Sevilla* en España, y ha sido incorporada a esta infraestructura gracias a la estrecha colaboración que se ha venido realizando con este grupo de investigación durante los últimos tres años.

Se trata de una plataforma basada en la tecnología *FPGA* que se utiliza para el estudio de la confiabilidad de sistemas digitales frente a *fallos transitorios* inducidos por radiación. El entorno permite configurar la *FPGA* con una versión real del sistema a estudiar e injectar fallos sobre éste, de forma no intrusiva, durante la ejecución de una aplicación. Los fallos se inyectan en el sistema a través de la emulación de fallos tipo *SEU* directamente sobre el sistema (emulación de *SEUs* por hardware). Los *SEUs* se emulan induciendo permutas del valor almacenado en ciertos registros por medio de reconfiguración parcial dinámica de la *FPGA*.

La plataforma está compuesta por una placa hardware de emulación de fallos basada en *FPGA* y un conjunto de herramientas software que permiten comprobar el diseño, recoger y analizar los resultados de las campañas de inyección de fallos.

La primera versión de *FT-Unshades* se basaba en dos instancias idénticas del circuito preparadas para ser utilizadas de forma concurrente. Estas instancias

son conocidas como módulos en prueba (*Module Under Test* (MUT)) y cada una tiene un nombre y función específica: *Target* y *Gold*. Con el fin de determinar el efecto que causa un fallo sobre el circuito, se emula un SEU en la instancia *target* y sus resultados se comparan con la instancia *gold* en cada ciclo de reloj (utilizando un módulo comparador - *COMP*). El esquema general del funcionamiento original de *FT-Unshades* se presenta en la Figura 4.18.

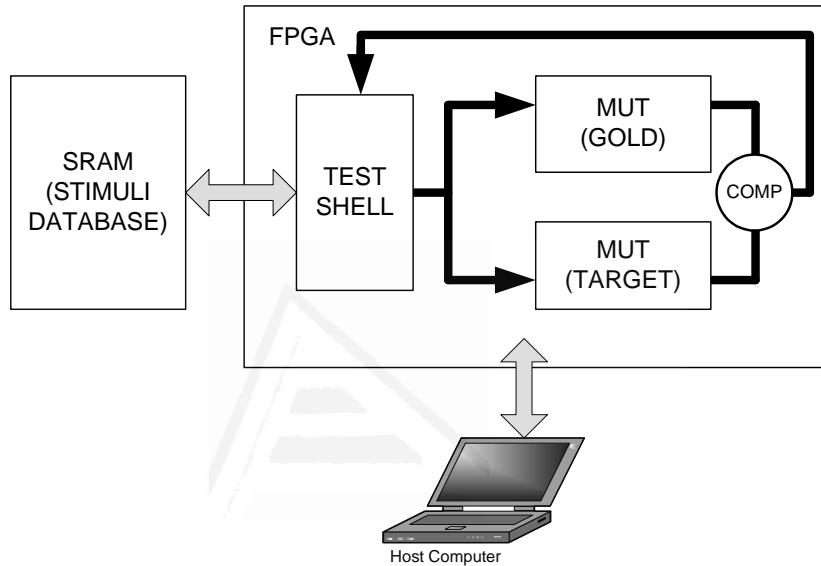


Figura 4.18: Esquema general del funcionamiento original de *FT-Unshades*

Como las técnicas software de recuperación de errores se basan en hacer comparaciones y re-calcular resultados para devolver al sistema a un estado libre de fallos, es posible que la instancia *target* (donde se inyectan los fallos) necesite ciclos de reloj adicionales para realizar la recuperación del fallo y obtener los resultados esperados (los mismos que los obtenidos por instancia *gold*). Este hecho origina que con el esquema original de *FT-Unshades*, que compara los resultados del circuito en cada ciclo de reloj, no sea posible analizar las bondades de las técnicas de endurecimiento software, ya que las dos instancias no pueden estar desalineadas en el tiempo. El sistema se basa en la coherencia ciclo a ciclo del estado de ejecución de sendos circuitos mediante la comparación de las salidas de interés.

El sistema se ha extendido para permitir el estudio de la confiabilidad de arquitecturas de microprocesador y de técnicas software de tolerancia a fallos. En [Guzmán-Miranda et al., 2009, Guzmán-Miranda, 2010] se puede encontrar una descripción exhaustiva de estas mejoras, las cuales se resumen a continuación. La idea principal del modelo de implementación mejorado es otorgar flexibilidad en el tiempo para que el circuito pueda recuperarse. En este caso, en lugar de instanciar dos copias del módulo en prueba (*gold* y *target*), el diseño

implementado tiene sólo una instancia del MUT (*target*), y la instancia *gold* se sustituye por una *Tabla Inteligente* o *Smart Table* (Figura 4.19).

La *Tabla Inteligente* es un autómata que implementa las restricciones relacionadas de tiempo necesarias para el test de sistemas basados microprocesador que implementan técnicas software de tolerancia a fallos. Es altamente flexible y configurable, para poder ser adaptado a diferentes arquitecturas de microprocesador. La *Tabla Inteligente* puede ser configurada en tiempo de emulación, es decir, después de la síntesis, implementación y configuración de la FPGA. En primer lugar, la *Tabla Inteligente* debe ser configurada con las salidas de una ejecución completa del circuito en donde no se inyecta ningún fallo, lo que significa que se debe emular un ciclo de trabajo completo del circuito bajo test, pero sin inyectar ningún *bit-flip*. Cuando está siendo configurada, la *Tabla Inteligente* no sólo memoriza la secuencia de salidas correctas, sino también los tiempos (medidos en ciclo de reloj) en los que las ocurren dichas salidas.

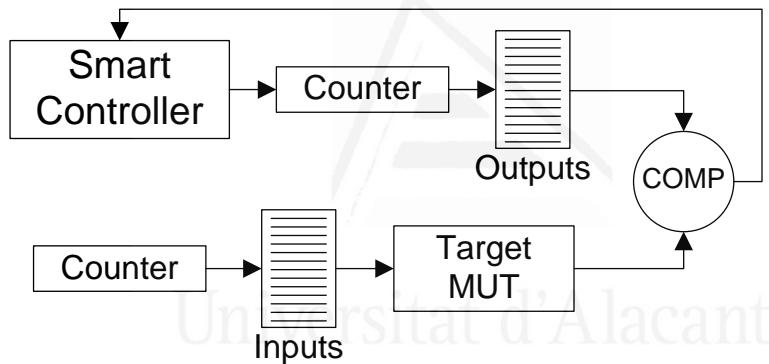


Figura 4.19: Esquema de funcionamiento de *FT-Unshades* usando la *Tabla Inteligente*

En este esquema de funcionamiento, tras configurar la *Tabla Inteligente* y llenarla con los pares *[Salida Esperada, Ciclo Reloj Esperado]*, el parámetro más importante que el usuario debe configurar es el *Tiempo de Recuperación Crítico* (T_{crit}), que es el tiempo máximo de recuperación, medido en ciclos de reloj, que se le permite al microprocesador. En otras palabras, si el microprocesador proporciona el valor correcto a la salida tras $Ciclo Reloj Esperado + T_{crit}$, el fallo se clasifica como incapaz de producir daño a la salida. Sin embargo, si han transcurrido $Ciclo Reloj Esperado + T_{crit} + 1$ ciclos de reloj y el microprocesador no ha obtenido aún la salida esperada, la *Tabla Inteligente* clasifica el fallo como *timeout*. Cabe notar que, ya que en ese momento la emulación se detiene, el *timeout* puede significar o bien que la ejecución del programa se ha congelado, o bien que el daño es tan crítico que la técnica de endurecimiento no ha podido recuperar los valores correctos antes de $T_{crit} + 1$ (es posible que necesite más tiempo). Además, debido a que se desea relajar las restricciones temporales del

test, pero sigue siendo necesario que la secuencia de salidas sea la correcta, si el microprocesador obtiene una salida incorrecta ($Salida \neq SalidaEsperada$) antes de $CicloRelojEsperado + T_{crit} + 1$, el fallo se clasifica como *daño*, es decir, el fallo ha modificado la salida válida esperada del programa.

Por otra parte, otra de las ventajas de *FT-Unshades* es que ofrece la posibilidad de realizar análisis jerárquicos del sistema, permitiendo realizar estudios de la confiabilidad del sistema por cada uno de sus módulos y submódulos. De esta forma, se evita la necesidad de realizar análisis sistemáticos para poder obtener información práctica. Es posible conocer información acerca del impacto que tiene en la confiabilidad del sistema cada uno de los módulos/submódulos que lo conforman [Aguirre et al., 2007]. Esta característica abre la posibilidad de aplicar y evaluar protecciones selectivas tanto en hardware como en software.

Por último, es importante mencionar también que esta herramienta permite tener acceso a todos los recursos de la micro-arquitectura del sistema. Incluso el alcance de inyección de la herramienta permite acceder a todos los posibles elementos de memoria significativos de un sistema microprocesador complejo, y a los bits de configuración de la FPGA.

En este trabajo, *FT-Unshades* ha sido incorporado a la infraestructura de endurecimiento presentada, y se ha utilizado para evaluar la confiabilidad proporcionada por cada una de las técnicas híbridas hardware/software desarrolladas. Al emular los fallos a nivel de hardware, y gracias a las ventajas de *FT-Unshades*, es posible obtener resultados más realistas que los obtenidos usando técnicas de inyección de fallos basadas en simulación, ya que en este caso es posible evaluar los diseños incluso a nivel de su microarquitectura. Por otra parte, a diferencia del ISS, con *FT-Unshades* es posible evaluar también los enfoques de endurecimiento hardware y los híbridos, no sólo las estrategias de protección basadas en software.

4.4. Conclusiones

En este capítulo se ha presentado una infraestructura que permite el diseño, implementación y evaluación de estrategias híbridas hardware/software de tolerancia a fallos. Esta infraestructura para el *co-endurecimiento* de sistemas embebidos está conformada principalmente por dos herramientas: un entorno para el endurecimiento software (*Software Hardening Environment - SHE*) y una herramienta para la evaluación de la confiabilidad (*FT-Unshades*).

El entorno desarrollado para el endurecimiento software (SHE) es el encargado de apoyar al diseñador y guiar el proceso de *co-endurecimiento*. Esta herramienta permite llevar a cabo las siguientes tareas:

- Diseñar técnicas software de tolerancia a fallos basadas en la redundancia de instrucciones a nivel de código ensamblador.
- Aplicar las técnicas diseñadas de forma automática en el código fuente de los programas.

- Realizar las tareas de endurecimiento para múltiples microprocesadores diferentes de forma transparente (de forma independiente de la arquitectura de microprocesador), gracias al *núcleo de endurecimiento genérico* que posee, el cual está basado en una *arquitectura genérica de microprocesadores*.
- Ofrecer una interfaz de programación de endurecimiento software (API) extensible que proporcione los métodos más habituales empleados en la programación de técnicas software de tolerancia a fallos.
- Facilitar el diseño de técnicas software que pueden ser aplicadas de forma completa o de forma selectiva a los diferentes recursos/componentes del programa, es decir, hace posible el diseño de técnicas software que se apliquen sólo a las partes/recursos más críticos.
- Obtener resultados preliminares (aunque muy realistas) acerca de la confiabilidad proporcionada por las técnicas diseñadas.

Por otro lado, *FT-Unshades* es la herramienta que se ha incorporado a la infraestructura para realizar la evaluación de la confiabilidad proporcionada por las estrategias híbridas de protección diseñadas. Está compuesta por una placa hardware de emulación de fallos basada en FPGA y un conjunto de herramientas software que permiten comprobar el diseño, recoger y analizar los resultados de las campañas de inyección de fallos. Sus principales ventajas son:

- Permite la inyección de fallos de forma no intrusiva, es decir, sin modificar el diseño original. Esto es posible ya que los fallos se emulan por medio de la reconfiguración parcial dinámica de la FPGA.
- Para realizar análisis sobre sistemas basados en microprocesador en los que se hayan implementado técnicas software de endurecimiento, *FT-Unshades* otorga flexibilidad en el tiempo para que el circuito en el cual se emulen los fallos pueda recuperarse y volver a un estado libre de errores (usando el esquema de funcionamiento de la *Tabla Inteligente*).
- Ofrece la posibilidad de realizar análisis jerárquicos, donde es posible estudiar por separado la confiabilidad de cada uno de los módulos internos que conforman el sistema.
- Al tratarse de una implementación real del sistema, se tiene acceso a todos los recursos físicos del mismo, incluso a nivel de la microarquitectura, lo cual facilita la obtención de resultados de confiabilidad más realistas que los que se pueden obtener por medio de técnicas de inyección de fallos basadas en simulación.
- Permite evaluar las técnicas de endurecimiento basadas en software y hardware.

El conjunto de las herramientas que conforman la infraestructura de endurecimiento permite realizar la exploración del espacio de diseño del *co-endurecimiento* de forma ágil y flexible. En el siguiente capítulo se presenta un

exhaustivo caso de estudio donde, utilizando la presente infraestructura para el *co-endurecimiento*, se valida de forma experimental el enfoque propuesto.



Universitat d'Alacant
Universidad de Alicante

Capítulo 5

Caso de estudio

El presente capítulo presenta un completo caso de estudio que valida la propuesta por medio de completa batería de pruebas y resultados experimentales. Para el caso de estudio ha sido seleccionado el microprocesador *PicoBlaze*, y se ha desarrollado un *front-end* y un *back-end* que integra dicho microprocesador dentro de SHE. Los experimentos realizados persiguen dos objetivos fundamentales: en primer lugar, se pretende verificar la flexibilidad y usabilidad del entorno de endurecimiento software, para lo cual se han diseñado y evaluado tres técnicas de endurecimiento software diferentes; en segundo lugar, se busca verificar la aplicabilidad del enfoque propuesto, objetivo para el cual se ha diseñado una estrategia de *co-endurecimiento hardware/software* que permite encontrar un punto óptimo en el espacio de diseño; la técnica desarrollada ha sido aplicada a tres aplicaciones distintas.

El capítulo está organizado de la siguiente forma: la Sección 5.1 presenta las características del microprocesador seleccionado para el caso de estudio; seguidamente, la Sección 5.2 describe el proceso de diseño y evaluación de las técnicas software de endurecimiento usando SHE; a continuación, la Sección 5.3 presenta de forma detallada algunos casos donde se aplica el *co-endurecimiento hardware/software* para el diseño de sistemas embebidos tolerantes a fallos; finalmente, en la Sección 5.4 se reúnen las conclusiones más importantes del presente capítulo.

5.1. Microprocesador: *Picoblaze*

Con el fin de evaluar la propuesta y demostrar la validez de la hipótesis de partida, se ha desarrollado un completo caso de estudio entorno al microprocesador *PicoBlaze* [XILINX, 2008]. Las principales razones que motivaron la selección de este microprocesador para este caso de estudio fueron las siguientes:

1. *PicoBlaze* es un *soft core* de 8 bits que posee severas limitaciones en cuanto a recursos disponibles, lo que hace imprescindible el co-diseño a medida

de la estrategia de mitigación de fallos para satisfacer los requisitos de las diferentes aplicaciones.

2. Es ampliamente utilizado y conocido por la industria en el diseño de sistemas embebidos basados en FPGA, lo que aumenta el interés de los resultados.
3. Al ser un *soft-core* es susceptible de ser portado a diferentes tipos de FPGAs y tecnologías ASIC. Además, ofrece la plasticidad necesaria para poder modificar los componentes internos del hardware en caso que sea necesario al aplicar alguna técnica de endurecimiento hardware. Los micróprocesadores COTS no cuentan, a priori, con dicha plasticidad, y en esos casos, deben ser endurecidos mediante técnicas software y componentes hardware redundantes externos al micróprocesador.
4. Sencillez, el *PicoBlaze* permite entender completamente la problemática que se quiere resolver, y facilita la exploración de técnicas de endurecimiento hardware/software, para posteriormente abordar micróprocesadores más complejos. Asimismo, gracias a esta sencillez será posible asumir los costes de fabricación de un ASIC (en tecnologías poco costosas, por ejemplo de 600 nm) para realizar ensayos reales de irradiación.

5.1.1. Características técnicas de *PicoBlaze*

Las características principales de este micróprocesador son:

- *Soft core* de 8 bits.
- Banco de registros con 16 registros de 8 bits de propósito general.
- Memoria de programa con capacidad para 1024 instrucciones (10 bits).
- ALU de 8 bits con banderas indicadoras de cero (*zero*) y acarreo (*carry*).
- Memoria *scratchpad* RAM de 64 bytes.
- 256 puertos de entrada y 256 puertos de salida.
- Pila de llamada a subrutinas de 31 posiciones.
- Rendimiento predecible, siempre 2 ciclos de reloj por instrucción.
- Ensamblador con sintaxis KCPSM3 [Chapman, 2003].

5.1.2. Integración de *PicoBlaze* con la infraestructura para el *co-endurecimiento*

Antes de realizar cualquier tarea propia del endurecimiento de sistemas, en este caso de estudio se ha realizado la integración del micróprocesador *PicoBlaze* con la infraestructura presentada en el capítulo anterior. Por un lado, para

integrar este procesador con SHE, se ha desarrollado un *front-end* y un *back-end* de compilación para dicho microprocesador. Además, en cuanto al hardware, se ha diseñado una versión RTL de este microprocesador que es equivalente a la versión original de *Xilinx*.

Front-end y back-end para PicoBlaze

Para llevar a cabo esta tarea, se han evaluado distintos compiladores de código abierto (*LANCE*, *LCC*, *GNU gcc*). Inicialmente se propuso desarrollar un nuevo *front-end* y un nuevo *back-end* para *PicoBlaze* en el compilador *GNU gcc* (que es uno de los más extendidos), para luego incorporar las características de tolerancia a fallos como una etapa intermedia dentro del proceso de compilación. Sin embargo, el problema con dicho compilador es que para implementar esta etapa intermedia de endurecimiento, era necesario conocer muy bien los lenguajes *GIMPLE* y *RTX*, con los que trabajan las capas intermedias (*middle-end*), los cuales son bastante complejos y poco documentados; además, durante el desarrollo del trabajo doctoral, las etapas de optimización internas de *GNU gcc* estaban siendo objetos de importantes reestructuraciones y cambios, lo que finalmente hubiera supuesto un trabajo obsoleto (más aún se observa que el proceso de cambios dentro de *GNU gcc* es muy profundo y durante los próximos meses continuará siendo objeto de discusión). Por este motivo, se ha replanteado la idea original de trabajar con *GNU gcc* y se han desarrollado desde cero los siguientes componentes:

- Un nuevo *front-end* a medida para *PicoBlaze*, que genera el flujo de instrucciones genéricas (GIF) para la arquitectura genérica que fundamenta SHE.
- Un *back-end* cuya labor principal es recibir el flujo de instrucciones genéricas endurecidas (HGIF) y trasladarlo nuevamente a la sintaxis del código ensamblador de *PicoBlaze*.

La herramienta seleccionada para llevar a cabo el desarrollo del *front-end* fue *ANTLR* (*ANother Tool for Language Recognition*) [Parr, 2011]. A continuación se mencionan las características más importantes de esta herramienta que motivaron su utilización:

- Compatible con gramáticas *LL(*)*.
- Generación automática de árboles de sintaxis abstracta — *Abstract Syntax Tree* (AST).
- Compatible con varios lenguajes de programación, entre los que destacan: *Java*, *C#*, *C*, *C++* y *Python*.
- Ideal para construir traductores e intérpretes para *DSL* (*Domain-Specific Languages*).

- Potente, moderna y muy sencilla de utilizar (mucho más fácil que las conocidas *Flex/Lex* y *Bison/Yacc*).
- Ofrece depuración de la gramática y la generación automática del código fuente a partir de la gramática definida.
- Ofrece un entorno integrado de desarrollo (*ANTLRWorks*) que permite visualizar las gramáticas de forma gráfica y además ofrece herramientas interesantes para la depuración de las gramáticas.
- Software libre.
- Muy documentada.

El *front-end* de *PicoBlaze* desarrollado para SHE cuenta con las siguientes características:

- Realiza análisis léxico, sintáctico y semántico del programa que se desea compilar.
- Genera el flujo de instrucciones genéricas (GIF) que es soportado por la arquitectura genérica propuesta.
- Utiliza la sintaxis *KCPSM3* para *PicoBlaze*.
- Multiplataforma.
- Basado en línea de comandos.

La Figura 5.1 representa el funcionamiento interno del *front-end*, del que se explican a continuación los componentes más importantes:

- El *front-end* recibe el código fuente de un programa en ensamblador (sintaxis *KCPSM3*).
- Realiza el análisis léxico del programa transformando el código fuente en un flujo de *Tokens* (identificadores, literales, operandos, palabras reservadas, puntuación). Además, los espacios en blanco (tabulaciones, espacios, caracteres de nueva línea) son descartados en este punto para no ser tenidos en cuenta dentro de los análisis siguientes. Esto hace que el código fuente pueda ser editado en sistemas operativos *Windows*, *Linux* y *MacOS* sin presentar incompatibilidades con respecto a los caracteres de retorno de carro y nueva línea.
- Analiza la sintaxis del programa (*parse*). Usando la secuencia de *Tokens* identificada en la etapa anterior, realiza el chequeo de la sintaxis (estructura) del programa con base a una serie de reglas establecidas en la gramática, y por último, genera el árbol sintáctico correspondiente a la estructura del programa, conocido como árbol AST.

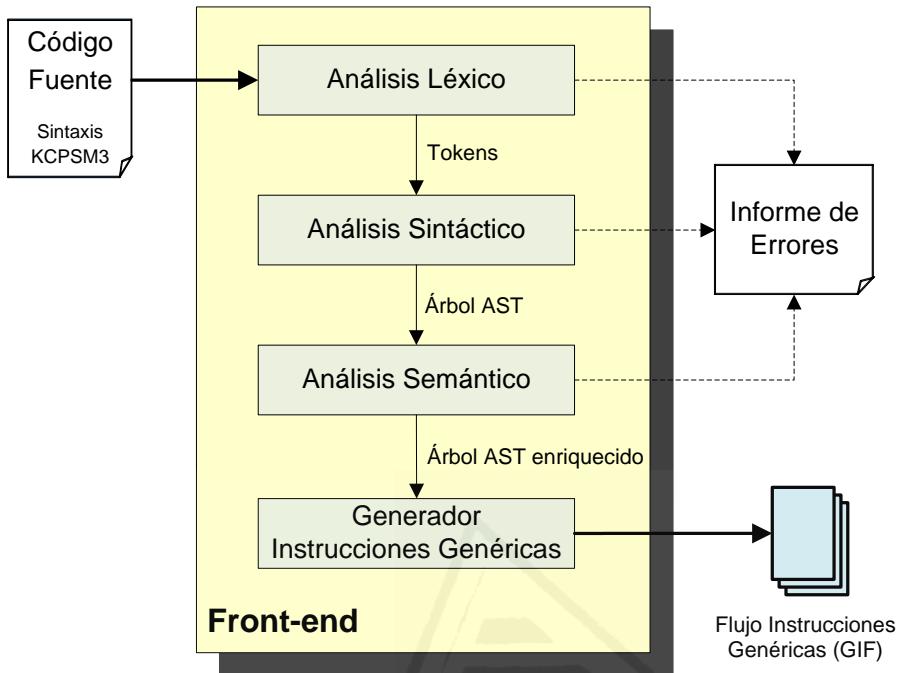


Figura 5.1: Funcionamiento interno del *front-end* para *PicoBlaze*

- Una vez realizado el análisis sintáctico, se realiza el análisis contextual (semántico) del programa, el árbol AST identificado es analizado para verificar si cumple con las restricciones semánticas del lenguaje, como por ejemplo: restricciones con el tipo de dato utilizado, tratar de reescribir una constante, entre otras. Finalmente, como resultado de esta fase, se genera un árbol AST enriquecido con la información obtenida durante el proceso de análisis semántico.
- Durante las etapas de análisis léxico, sintáctico y semántico, es posible crear un informe de errores.
- El árbol AST enriquecido se pasa al generador de instrucciones genéricas, que como su nombre indica, genera una a una las instrucciones genéricas y las ensambla dentro de la arquitectura genérica diseñada.
- El *front-end* entrega un flujo de instrucciones genéricas ensambladas en la arquitectura genérica propuesta. El proceso en adelante es responsabilidad de las herramientas disponibles en el *núcleo de endurecimiento genérico* (*endurecedor e ISS*).

En la Figura 5.2 se presenta un ejemplo de la salida del *front-end* desarrollado en este trabajo cuando éste detecta un error. La salida está comparada con la

salida del compilador KCPSM3 de la empresa *Xilinx*, donde se puede apreciar el mayor nivel de detalle que se obtiene con nuestro compilador. Por ejemplo, en la figura se presenta un programa que contiene tres errores, dos de ellos semánticos (líneas 27 y 29) y uno sintáctico (línea 33) y se observa la respuesta de los dos compiladores.

La imagen muestra una comparación entre el front-end desarrollado por UniCAD y el compilador Xilinx KCPSM3. Se presentan dos capturas de pantalla:

- UniCAD:** Una terminal de línea de comandos que muestra el resultado de la compilación de un programa. Los errores detectados son:
 - Línea 27: Semantic error at 27:15 : The identifier 's80' is not an aliased register.
 - Línea 29: Semantic error at 29:15 : The identifier 'FF1' is not an aliased register.
 - Línea 33: Syntaxic errors found. It is not possible to build the AST tree.
- Xilinx KCPSM3:** Una terminal de línea de comandos que muestra el resultado de la compilación. Los errores detectados son:
 - Línea 33: ERROR - No second operand specified for COMPARE instruction. Please correct and try again.

Los errores de UniCAD están resaltados con círculos rojos y tienen una flecha que apunta a la terminal de KCPSM3, indicando que ambos detectaron el mismo problema.

Figura 5.2: Comparativa del informe de errores del *front-end* desarrollado versus el de *Xilinx KCPSM3*

Además, merece la pena resaltar que la herramienta de la empresa *Xilinx* para ensamblar el código fuente de los programas sólo funciona en la familia de sistemas operativos *Windows*, mientras que el *front-end* desarrollado para este caso de estudio es multiplataforma.

Para evaluar la funcionalidad y corrección del *front-end* se ha decidido utilizar el enfoque de pruebas de regresión, lo que ha permitido comenzar la evaluación del software en paralelo con el desarrollo del sistema. Para ello, se ha definido un procedimiento automático encargado de realizar la compilación de un conjunto de programas de prueba y se han predefinido los resultados esperados de la compilación para cada uno de ellos. El test de regresión comprobará que los resultados sean los esperados para cada uno de los programas de prueba. Para cada nueva iteración en el desarrollo del compilador, el test de regresión verifica la corrección de las modificaciones introducidas. Finalmente, el procedimiento de prueba de regresión, presenta un informe con el número y porcentaje de programas de prueba que han obtenido el resultado esperado y han pasado satisfactoriamente la prueba. El nivel de calidad esperado por ningún motivo puede ser inferior al 100 %. Con esto se obliga a que las futuras versiones del compilador, además de introducir nuevas funcionalidades, sigan soportando correctamente todas las desarrolladas hasta el momento.

Los programas se han ido incorporando al banco de pruebas paulatinamente, a medida que se iban desarrollando nuevas funcionalidades o cuando se descubría y se corría algún tipo de problema o caso especial en alguna de

las etapas del proceso de compilación. Al concluir el desarrollo del *front-end*, el banco de pruebas quedó conformado por un total de 477 programas, entre los cuales se pueden encontrar programas de las siguientes características:

- Programas sencillos donde se verifica el funcionamiento de las reglas léxicas, sintácticas y semánticas del compilador, establecidas para cada una de las instrucciones del ensamblador de *PicoBlaze* (sintaxis KCPSM3).
- Programas para probar características especiales del compilador, propias de este trabajo de investigación, como la incorporación de algunos *pragmas* del compilador.
- Programas reales publicados en Internet por la comunidad científica y de desarrollo de *PicoBlaze*.

Es importante aclarar que cada programa de prueba puede ser completamente correcto o contener errores identificados (léxicos, sintáticos o semánticos). De esta forma la prueba de regresión permite evaluar no sólo si el compilador es capaz de encontrar todos los errores, sino también si es capaz de clasificarlos de forma adecuada.

Para automatizar el proceso de pruebas de regresión se ha realizado la implementación de dos tareas: en primer lugar, el desarrollo de un *pragma* para indicarle al compilador el número de errores de cada tipo (léxico, sintáctico y semántico) que debe esperar al realizar la compilación de cada programa de prueba; la segunda tarea, consistió en el desarrollo de una opción adicional del compilador que permite ejecutar de forma automática los tests de regresión, la cual realiza la compilación de todo el banco de pruebas, verificando la coincidencia de errores encontrados con errores esperados y por último, presentar un informe donde se resumen los resultados de toda la prueba.

El *pragma REGRE* sirve para que el compilador conozca los errores de cada tipo que debe esperar para cada uno de los programas de prueba. Esto se logra cuando a cada programa de prueba se le escribe al final de su código fuente dicha directiva del compilador. El *pragma* tiene la siguiente forma:

```
//REGRE LexErr SinErr SemErr
```

La palabra reservada `//REGRE` y a continuación tres campos variables (números enteros) que indican el número de errores esperado de cada tipo: `LexErr` es el número de errores de tipo léxico, `SinErr` es el número de errores sintácticos, y `SemErr` es el número de errores de tipo semántico. Por ejemplo, la siguiente sentencia en un programa de prueba:

```
//REGRE 0 2 5
```

Indica que el compilador debe esperar encontrar en ese programa de prueba: 0 errores léxicos, 2 errores sintácticos y 5 errores semánticos.

La opción del compilador para ejecutar todo el test de regresión (`-t` o bien `-runregressions [=VALUE]`), realiza la compilación de todo el banco de

pruebas almacenado en el directorio VALUE, verificando que se obtengan los errores esperados. Al ejecutar esta opción, se obtiene como salida, para cada uno de los programas de prueba la información que se presenta a continuación:

- Nombre del programa de prueba.
- En caso que el archivo contenga errores, la lista completa de los errores, junto con el tipo y descripción del error.
- Una línea de resumen, donde se presentan el número de errores de cada tipo encontrados, acompañados de la palabra *PASSED!* en caso que la prueba haya sido exitosa, y *NOT PASSED!* en caso contrario, indicando el número de errores de cada tipo que eran esperados.

Por último, se presenta también un informe final donde se resumen los resultados de toda la prueba. En la Figura 5.3 se presenta una captura de pantalla de parte de la salida generada por la ejecución de la prueba de regresión.

La Tabla 5.1 presenta los resultados obtenidos de la ejecución de la prueba de regresión para realizar la evaluación funcional del *front-end* desarrollado. Por tratarse de una prueba para verificar la funcionalidad del compilador, el resultado del comportamiento esperado del programa debía ser el 100 %. Si en algún momento, debido a un cambio en el programa o nuevas funcionalidades implementadas, dicho porcentaje disminuye, esto quiere decir que se ha visto afectada alguna de las funcionalidades que operaban correctamente hasta este momento en el compilador; y por consiguiente, el problema debe ser corregido.

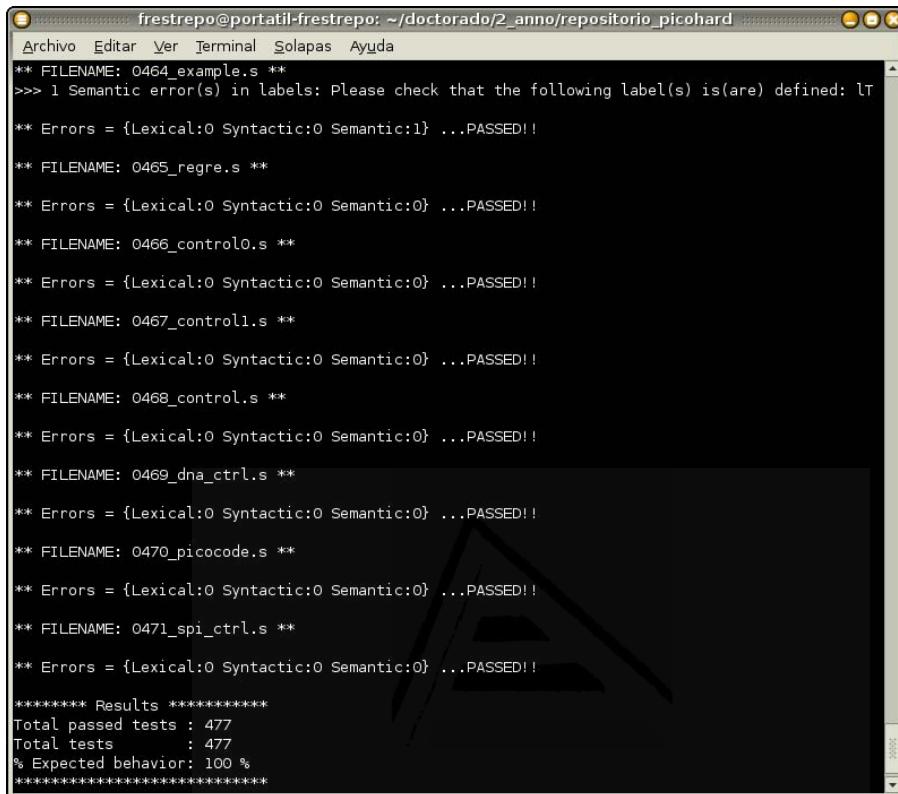
Resultados de la prueba de regresión	
Programas de prueba	477
Programas que pasaron la prueba	477
Porcentaje comportamiento esperado	100 %

Cuadro 5.1: Resultados obtenidos en la prueba de regresión

RTL *PicoBlaze*

A la hora de escoger el IP del microprocesador, se evaluaron distintas alternativas. Por un lado, las versiones oficiales de *PicoBlaze* que ofrece el fabricante *Xilinx* [XILINX, 2008] están optimizadas mediante la utilización de primitivas propias de sus arquitecturas FPGA, lo que limita su portabilidad. Por otro lado, existen versiones alternativas, que si bien son ISA compatibles, no respetan su microarquitectura [Bleyer-Kocik P., 2011].

Para este caso de estudio se optó por desarrollar una nueva versión que fuera RTL equivalente a la original (*RTL PicoBlaze*) y totalmente portable. De esta forma, el *RTL PicoBlaze* permite validar la propuesta de *co-endurecimiento* no sólo para FPGAs de cualquier fabricante, sino también para diferentes tecnologías de ASIC.



```
frestrepo@portatil-frestrepo: ~/doctorado/2_anno/repositorio_picohard
Archivo Editar Ver Terminal Solapas Ayuda
** FILENAME: 0464_example.s **
>>> 1 Semantic error(s) in labels: Please check that the following label(s) is(are) defined: lt
** Errors = {Lexical:0 Syntactic:0 Semantic:1} ...PASSED!!

** FILENAME: 0465_regres.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0466_control0.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0467_control1.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0468_control.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0469_dna_ctrl.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0470_picocode.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

** FILENAME: 0471_spi_ctrl.s **

** Errors = {Lexical:0 Syntactic:0 Semantic:0} ...PASSED!!

***** Results *****
Total passed tests : 477
Total tests       : 477
% Expected behavior: 100 %
*****
```

Figura 5.3: Captura de pantalla del test de regresión para verificar la funcionalidad del *front-end*

5.2. Desarrollo de técnicas de endurecimiento software usando SHE

Como parte de este caso de estudio, se han desarrollado tres técnicas software de tolerancia a fallos usando el entorno de endurecimiento software presentado (SHE). El diseño y evaluación de dichas técnicas busca validar las siguientes características de SHE:

- La facilidad para desarrollar técnicas software de tolerancia a fallos mediante un API de endurecimiento. Las técnicas están basadas en reglas de transformación del código fuente de los programas (a nivel de ensamblador), lo cual permite que puedan ser aplicadas de forma automática.
- La relevancia de la información ofrecida, tanto por el *endurecedor* como por el simulador del repertorio de instrucciones (ISS), para guiar el co-diseño de la estrategia de endurecimiento.

- La precisión del *ISS* para evaluar la confiabilidad del software, y de esta forma, poder valorar las mejoras de confiabilidad que se obtienen al aplicar las técnicas diseñadas.

Las técnicas de endurecimiento desarrolladas han tenido en cuenta todos los criterios anteriores, y no solamente los niveles de confiabilidad que éstas podían ofrecer.

5.2.1. Diseño de técnicas software

Se han diseñado tres técnicas software de mitigación basadas en la utilización de redundancia de instrucciones a nivel de código ensamblador, denominadas: *TMR1*, *TMR2* y *SWIFT-R*¹. Las tres se basan en la conocida técnica de la triple redundancia modular (en inglés *Triple Modular Redundancy* — TMR) [Von-Neumann, 1956]. A continuación se explica en detalle cada una de ellas.

Técnica *TMR1*

Se trata de una adaptación software de la técnica TMR que se aplica en función del tipo de instrucciones. Los pasos necesarios para endurecer el código fuente de un programa usando esta técnica pueden resumirse como se presenta a continuación:

1. El primer paso es la identificación de los *bloques básicos* (nodos del grafo de control de flujo de ejecución — CFG) que conforman el programa. Aquí se incluye también la identificación de los *sub-nodos* del CFG.
2. A continuación se construye el CFG del programa.
3. Se triplican las instrucciones que haya decidido el diseñador, por ejemplo: instrucciones aritméticas, lógicas, desplazamientos, etc. Para triplicar una instrucción, se inserta la instrucción original y se hacen dos copias adicionales. Las instrucciones redundantes deben operar usando copias de los registros. Si se detecta que no hay copias de algún registro para operar en la instrucción redundante, se deben insertar instrucciones para hacer las copias de los registros antes de triplicar la instrucción.
4. Se insertan votadores por mayoría seguidos de procedimientos de recuperación para los registros protegidos. Esta inserción ocurre únicamente en los siguientes puntos críticos:
 - Antes de la última instrucción de cada nodo/sub-nodo del CFG.
 - Antes de cualquier instrucción que sea el destino de un salto o llamada a subrutina.

¹La versión de *SWIFT-R* presentada aquí es una adaptación de la técnica general de recuperación de errores propuesta por [Reis et al., 2007] con el mismo nombre.

5. Las copias de los registros deben liberarse después de que se haya verificado/recuperado el valor correcto, es decir, después de cada ejecución del votador por mayoría y procedimiento de recuperación. Esto sirve para liberar recursos que pueden ser utilizados para continuar añadiendo redundancia en otras partes del código fuente.
6. En el caso de que no se tengan suficientes registros para seguir haciendo copias, se deben insertar más votadores y procedimientos por votación de forma dinámica. De esta manera, se verifican los valores correctos y se liberan las copias de los registros que tengan réplicas, y así se obtienen los recursos necesarios para continuar con el proceso de endurecimiento.

Nótese que el diseñador tiene la flexibilidad para elegir cuales son los tipos de instrucciones que desea triplicar (ver Figura 4.5). De esta forma, se puede ajustar la protección aplicada a los programas de acuerdo a si llevan un uso intensivo de operaciones aritméticas o de otro tipo. Además puede combinar la protección de varios tipos de instrucciones, por ejemplo seleccionando que se protejan las instrucciones de tipo aritmético, lógico y de desplazamientos al mismo tiempo. La decisión acerca de cuales instrucciones deben ser protegidas dependerá de factores como: el tipo de programa, el impacto que se causa sobre el tamaño del código y el rendimiento al aplicar la técnica, y la confiabilidad obtenida para cada caso.

La Figura 5.4 presenta una versión simplificada de la implementación de la técnica *TMR1* mediante el API de endurecimiento disponible en el *núcleo de endurecimiento genérico* de SHE.

Para mayor claridad, en la Figura 5.5 se presenta un ejemplo de un programa endurecido mediante la técnica *TMR1* aplicada a las instrucciones aritméticas. El mismo programa de ejemplo será utilizado en ejemplos posteriores para poder comparar las semejanzas y diferencias más importantes con la aplicación de otras técnicas diseñadas (*TMR2* y *SWIFT-R*).

Técnica *TMR2*

Consiste en detectar y corregir fallos en los datos del programa mediante el computo doble de todas las operaciones, y sólo en caso de que exista alguna discrepancia entre los dos primeros resultados, se re-calcula la operación una tercera vez para corregir el error.

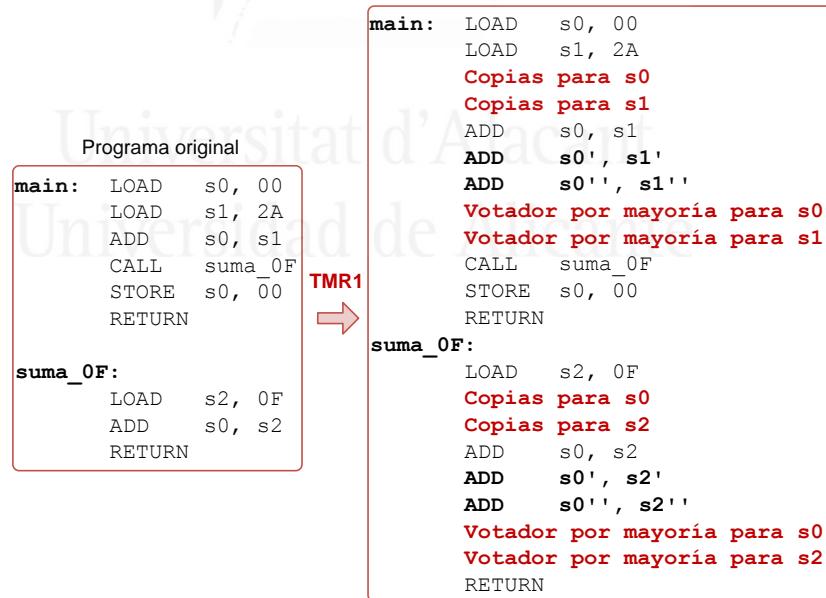
Nótese que para aplicar esta técnica no se requiere conocer el CFG de los programas (como en el caso de *TMR1*), basta con hacer transformaciones directas al código de manera automática. A diferencia de *TMR1* donde las comprobaciones (votadores y procedimientos de recuperación) se insertan únicamente en los puntos críticos del CFG; en *TMR2*, inmediatamente después de cada operación, ésta se duplica y se comprueba su resultado.

Por otra parte, *TMR2* también puede ser aplicada de forma selectiva e incremental, ya que el diseñador puede elegir cuales son los tipos de instrucción a los cuales se les aplica la transformación (aritméticas, lógicas, desplazamientos, rotaciones, ...).

```

public static bool TMR1Hard(MicroGenArch mgaOri, MicroGenArch mgaHard,
    List<GenInstType> lstTypesToHard, ...)
{
    foreach (GenIns gi in mgaOri.program.GIF)
    {
        if((lstTypesToHard.Contains(gi.type)) && (gi.hStuff.hardenable))
        {
            neededRegs = NumberNeededRegisters(...) // Num de registros necesarios
            if(neededRegs > mgaOri.NumberAvailableRegisters()) // > Num de registros disponibles
                InsertRecoverProcAndFreeResources(mgaOri, mgaHard, voter, ...) // inserta votador
                // por mayoría, proc de recuperación y libera recursos
            if(item.hStuff.hardenable)
            {
                CopyNeededRegisters(...) // Hace copias de los registros necesarios
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
                RedundateInstruction(mgaOri, mgaHard, 2, gi, ...) // Triplica instrucción
            }
            else
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
            if(...cambio nodo/subnodo...)
                InsertRecoverProcAndFreeResources(mgaOri, mgaHard, voter, ...)
        }
        else
        {
            if(...cambio nodo/subnodo...)
                InsertRecoverProcAndFreeResources(mgaOri, mgaHard, voter, ...)
            mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
        }
    }
    return true
}

```

Figura 5.4: Implementación de *TMR1* usando el API de endurecimientoFigura 5.5: Programa de ejemplo endurecido con *TMR1*

En la Figura 5.6 se muestra la versión simplificada de la implementación de

la técnica TMR2 usando el API de endurecimiento.

```

public static bool TMR2Hard(MicroGenArch mgaOri, MicroGenArch mgaHard,
                           List<GenInstType> lstTypesToHard, ...)
{
    foreach (GenIns gi in mgaOri.program.GIF)
    {
        if((lstTypesToHard.Contains(gi.type)) && (gi.hStuff.hardenable))
        {
            if(NumberNeededRegisters(...) > mgaOri.NumberAvailableRegisters()) // Sin recursos
            {
                gi.hStuff.hardenable = false;
                gi.hStuff.toolMsg = "Not enough resources to hard this instruction";
            }
            if(gi.hStuff.hardenable)
            {
                CopyNeededRegisters(...) // Hace copias de los registros necesarios
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
                RedundateInstruction(mgaOri, mgaHard, 1, gi, ...) // Duplica la instrucción
                InsertDetectionProc(mgaOri, mgaHard, comparision, ...) // Comparador
                RedundateInstruction(mgaOri, mgaHard, 1, gi, ...) // Triplicado de la instrucción
                InsertRecoverProc(mgaOri, mgaHard, ...) // Procedimiento de recuperación
                mgaOri.ReleaseRegisterCopies() // Libera copias
            }
            else
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
        }
        else
            mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
    }
    return true
}

```

Figura 5.6: Implementación de TMR2 usando el API de endurecimiento

La Figura 5.7 presenta el mismo programa de ejemplo que antes, pero esta vez endurecido mediante la técnica TMR2.

Técnica SWIFT-R

Se trata de un método general de recuperación de errores en la sección de datos de los microprocesadores. Esta técnica fue propuesta inicialmente por [Reis et al., 2007], y en este caso de estudio se ha realizado una adaptación de la misma donde se proponen mejoras para aumentar su flexibilidad, ya que es posible aplicarla de forma selectiva en los programas. Puede resumirse en los siguientes pasos:

1. Como en TMR1, se identifican los *bloques básicos* del programa, que permiten identificar los nodos y subnodos del CFG. Se presta especial atención a la clasificación de las instrucciones de acuerdo al tipo de flujo de datos que ocasionan con respecto a la *esfera de replicación* — SoR (ver la Sección 4.2.3). Merece la pena recordar que las instrucciones que generan un flujo de datos desde dentro hacia afuera de la esfera (instrucciones *outSoR*) delimitan la subdivisión de los nodos en subnodos.
2. Construcción del CFG del programa.

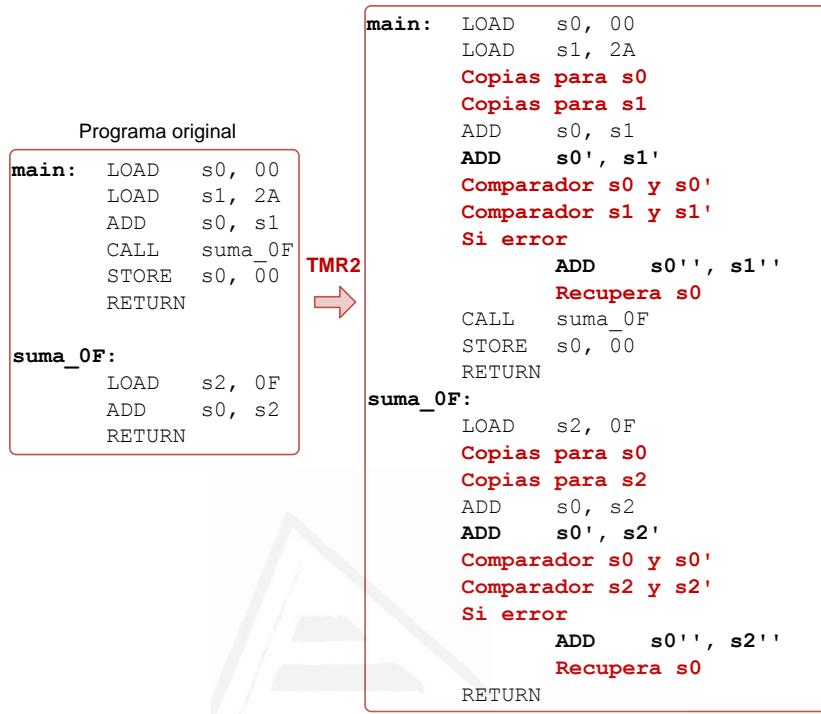


Figura 5.7: Programa de ejemplo endurecido con TMR2

3. Triplicación de los datos que entran por primera vez dentro de la SoR, es decir, los datos implicados en la ejecución de instrucciones *inSoR*. En este caso de estudio se considera que sólo el banco de registros del microprocesador está incluido dentro de la SoR, mientras que el subsistema de memoria se considera fuera de la esfera, ya que se asume que la memoria tiene sus propios mecanismos de protección. Por lo tanto, en este caso, para cada instrucción clasificada como *inSoR* (lectura de un puerto de entrada, lectura de un dato almacenado en la memoria, carga de un literal en un registro), se deben crear dos copias adicionales del dato que ingresa en la esfera. Estas copias redundantes deben hacerse simplemente copiando el valor del registro, sin repetir los accesos a memoria o las lecturas desde los puertos de entrada.
4. Se triplican las instrucciones que realizan alguna operación con los datos (instrucciones aritméticas, lógicas, desplazamientos, rotaciones). Las instrucciones redundantes deben operar con las copias de los registros.
5. Se verifica la consistencia de los datos por medio de la inserción de votadores por mayoría y procedimientos de recuperación. Esta verificación debe hacerse únicamente antes de la ejecución de las siguientes instruc-

ciones:

- Instrucciones clasificadas como *outSoR*, en este caso: almacenamientos de datos en memoria, y escrituras en los puertos de salida.
 - Las instrucciones localizadas justo antes de un salto condicional. Esta verificación es necesaria porque dichas instrucciones pueden afectar los resultados de las banderas de la ALU (*zero*, *carry*, ...), entonces si el valor almacenado en un registro está corrupto, la bandera resultante de la operación puede verse afectada también, y por consiguiente, se puede ocasionar un salto incorrecto a alguna parte del CFG.
6. Las copias de los registros pueden liberarse únicamente si estos registros no se utilizan más en la parte restante del programa, de lo contrario, las copias deben mantenerse todo el tiempo desde su creación. Esta condición implica realizar un análisis detallado del CFG.

La Figura 5.8 presenta la implementación de esta técnica (versión simplificada) usando el API de endurecimiento.

```
public static bool SWIFT_R(MicroGenArch mgaOri, MicroGenArch mgaHard, ...)
{
    if(...selective SWIFT-R...)
        RecalculateSoR(ref mgaOri, regsToHarden)
    mgaOri.program.CalculateNodes()
    mgaOri.program.ExtractControlFlowGraph()
    foreach (GenIns gi in mgaOri.program.GenInsFlow)
    {
        switch(gi.hStuff.sorType)
        {
            case SoRinsType.IN:
            {
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
                RedundateSoRin(mgaOri, mgaHard, gi,...) // Triplica los datos
            }
            case SoRinsType.NORMAL:
            {
                mgaHard.InsertInstruction(gi, ...) // Inserta la instrucción original
                RedundateInstruction(mgaOri, mgaHard, gi,...)// Triplica instrucción
            }
            case SoRinsType.OUT:
            {
                InsertRecoverProc(mgaOri, mgaHard, voter,...) // Inserta votador
                mgaHard.InsertInstruction(gi,...) // Inserta la instrucción original
            }
        }
    }
    return true
}
```

Figura 5.8: Implementación de SWIFT-R usando el API de endurecimiento

En la Figura 5.9 se muestra el mismo programa de ejemplo que para las dos técnicas anteriores, esta vez endurecido mediante la técnica *SWIFT-R*.

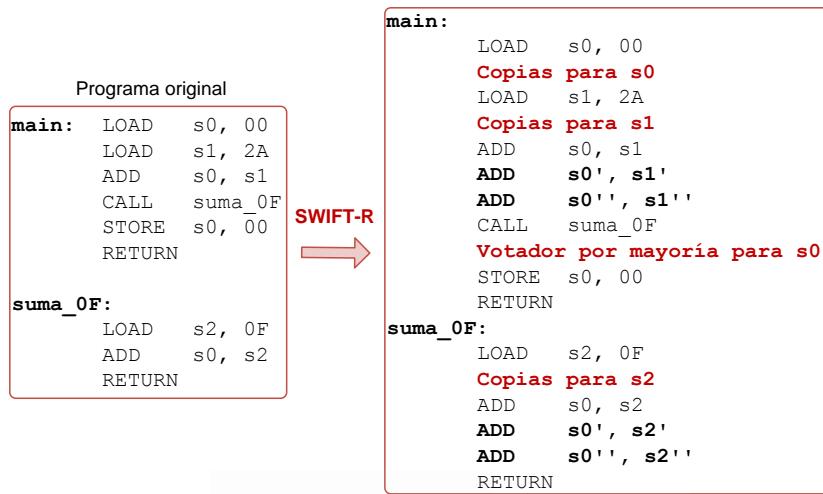


Figura 5.9: Programa de ejemplo endurecido con *SWIFT-R*

Esta implementación de *SWIFT-R*, supone una mejora respecto de la original, ya que brinda al diseñador la posibilidad de seleccionar subconjuntos de registros del banco de registros para ser endurecidos. Esto es posible gracias a la implementación flexible de la esfera de replicación (SoR) dentro de SHE. Básicamente, el nuevo enfoque consiste en trasladar los registros que no se desean endurecer hacia afuera de la SoR, y de esta forma, las transformaciones realizadas en el código se encargarán de proteger solamente los registros que permanezcan dentro de la esfera. Esta característica permite realizar una exploración de grano fino del espacio de diseño en términos de confiabilidad, impacto sobre el tamaño del código fuente, y degradación del rendimiento. En la Sección 5.3 se presentan con más detalle las características de endurecimiento selectivo implementadas en *SWIFT-R*.

Discusión preliminar

Nótese que las técnicas TMR1 y TMR2 tratan las instrucciones a proteger como si fueran estructuras hardware, es decir, estas técnicas son adaptaciones muy cercanas al funcionamiento de TMR en hardware. TMR1 inserta votadores por mayoría y procedimientos de recuperación justo antes de abandonar cada nodo/subnodo del CFG, mientras que TMR2 lo hace después de cada instrucción. Esta estrategia protege los resultados generados por un nodo/subnodo o por una instrucción, según el caso. Sin embargo, este hecho también ocasiona un incremento del tiempo de ejecución, porque se necesitan más ciclos de reloj. Tiempo éste en el que el registro original es vulnerable a fallos transitorios inducidos por radiación que potencialmente pueden llegar a causar un error en el sistema. Esta situación indica que los resultados previstos de confiabilidad obtenidos gracias a las técnicas TMR1 y TMR2 no serán muy satisfactorios, da-

do que el compromiso entre el endurecimiento que se aplica y el incremento en el tiempo de ejecución producido limita la cobertura efectiva frente a fallos.

Por el contrario, en el caso de *SWIFT-R*, a diferencia de los otros dos enfoques, las copias de los registros se mantienen durante todo el tiempo de vida de los registros, ya que sólo se liberan los recursos redundantes en el caso de que no sean utilizados en la parte restante del programa. De esta forma, las copias redundantes se mantienen incluso entre los saltos entre diferentes nodos del CFG, y esto facilita que no haya tantos puntos críticos en los que pueda haber fallos. Se espera que los resultados de confiabilidad de *SWIFT-R* sean mejores que los de *TMR1* y *TMR2*.

5.2.2. Evaluación de técnicas software

Para realizar la evaluación de las técnicas software es necesario contar con un banco de programas de prueba (*test benchmark*) sobre el cual se puedan aplicar las técnicas y evaluar los resultados obtenidos. Debido a las limitaciones inherentes del procesador *PicoBlaze* en términos de tamaño de memoria (1024 instrucciones en la memoria de programa y 64 bytes en la de datos), no es factible la utilización de un *benchmark* estándar para sistemas embebidos, como por ejemplo: *MiBench* [Guthaus et al., 2001], o *EEMBC* [EEMBC, 2011]. Para este trabajo, se desarrolló uno propio, adaptado a las limitaciones particulares de *PicoBlaze*.

El *benchmark* está compuesto por programas típicos de microprocesadores/microcontroladores de 8 bits. Incluso la mayoría de estos programas están presentes normalmente dentro de los bancos de prueba estándar y son utilizados por otros autores de la comunidad científica para la evaluación de técnicas de endurecimiento software. Los programas han sido implementados usando la sintaxis *KCPSM3* para el microprocesador *PicoBlaze*. El banco de pruebas está compuesto por los siguientes doce (12) programas, ocho (8) básicos y cuatro (4) complejos:

- Básicos:

1. Método de ordenamiento *burbuja* (*bub*).
2. División escalar (*div*).
3. *Fibonacci* (*fib*).
4. Máximo común divisor (*gcd*).
5. Suma de matrices (*madd*).
6. Multiplicación escalar (*mult*).
7. Potenciación (*pow*).
8. Método de ordenamiento *quick sort* (*qsort*).

- Complejos:

1. *Advanced Encryption Standard* (*aes*) o *Rijndael*.

2. Filtro *Finite Impulse Response (fir)*.
3. Multiplicación de matrices (*mmult*).
4. Controlador *Proportional Integral Derivative (pid)*.

Todos los programas han sido endurecidos, con cada una de las tres técnicas diseñadas, de forma automática usando SHE. Para el caso de *TMR1* y *TMR2* los tipos de instrucciones seleccionados para ser protegidos han sido las instrucciones aritméticas y las lógicas. Esta decisión fue tomada principalmente para permitir que las versiones endurecidas se ajustaran al limitado tamaño de memoria de programa de *PicoBlaze*. En el caso de *SWIFT-R*, ha sido aplicada a todo el banco de registros.

La evaluación de las técnicas se ha realizado considerando los siguientes aspectos:

- Equivalencia funcional de los programas endurecidos con respecto a los originales (no endurecidos).
- Impacto del endurecimiento sobre el tamaño del código fuente y el rendimiento de los programas (*overheads*).
- Nivel de confiabilidad.

A continuación se presentan los resultados obtenidos.

Verificación funcional

El objetivo de esta parte de la evaluación del sistema de endurecimiento persigue verificar la correcta funcionalidad del *endurecedor* y de las técnicas software diseñadas; y no su capacidad de endurecimiento. Se pretende validar si los programas endurecidos son funcionalmente equivalentes a los programas originales, o si por el contrario, durante el proceso de endurecimiento se ha hecho alguna modificación no deseada en el comportamiento de los programas.

Para realizar dicha verificación, ha sido necesaria la implementación de un *pragma* en el compilador. Este *pragma* permite informar al simulador cuales son los resultados finales que se espera obtener tras la simulación del programa, en cuanto al estado final de sus registros y memoria. Esta directiva puede tener cualquiera de las dos formas siguientes (que no son excluyentes):

```
; Output [ADDR] = n1, n2, n3, ...
; Output Reg1 = val1, Reg2 = val2, ...
```

La primera sirve para especificar los resultados esperados que se almacenan en memoria, donde los n_i son los valores esperados que se almacenan a partir de la posición de memoria *ADDR*. Por otra parte, la segunda forma de la directiva de compilación sirve para determinar el valor esperado en el banco de registros del microprocesador. Es importante aclarar que estas directivas pueden

aparecer en el código fuente de un programa en repetidas ocasiones, y que todos los números están representados en el sistema de numeración hexadecimal (00 – FF). Por ejemplo:

```
;Output [00] = 00, 0F, 3F, 05  
;Output [1A] = 10, AA  
;Output s0 = 01, sF = B2
```

Significa que en las posiciones de memoria 00, 01, 02, 03, 1A, 1B se esperan los valores 00, 0F, 3F, 05, 10, AA respectivamente, y en los registros s0 y sF se esperan los valores 01 y B2 respectivamente.

Haciendo uso de esta directiva del compilador (`Output`), como se mencionó en la Sección 4.2.2, se han desarrollado funcionalidades en el ISS que permiten realizar verificaciones especiales:

- **Verificar resultados esperados (`-c`, `--check`)**. Permite verificar el correcto funcionamiento del programa simulado al permitir comprobar si el estado final de la memoria y/o registros, una vez concluida la simulación, coincide con los resultados esperados. Además, es posible verificar también la secuencia de salidas que ha entregado el programa en los diferentes puertos de salida del microprocesador. El ISS conoce cuales deben ser los resultados obtenidos (gracias al *pragma Output*). En la Figura 5.10 se pueden observar los resultados de la simulación de un programa de prueba al cual le ha sido incorporada la directiva de comprobación y ha encontrado exitosamente los resultados esperados, mientras que en la Figura 5.11 se presenta el mismo ejemplo pero en el caso en el que la verificación funcional no haya sido exitosa.
- **Verificar resultados esperados del programa original en el programa endurecido (`-c`, `--check-hardening [=VALUE]`)**. Esta segunda opción es similar a la anterior, pero de forma adicional realiza las mismas comprobaciones sobre el programa endurecido. *VALUE* es el nombre del programa endurecido. Esta opción es útil para verificar que no se ha modificado la funcionalidad del programa original durante el proceso de endurecimiento y los resultados finales del programa endurecido concuerdan con los resultados originales. Por tanto, con esta opción, se llevan a cabo dos simulaciones, una sobre el código fuente original y la otra sobre su dual endurecido. La Figura 5.12 presenta un ejemplo de los resultados de la simulación comprobando que el programa original y el endurecido obtienen los mismos resultados esperados (indicados mediante el uso de la directiva del compilador `Output` en el código fuente del programa original).

Todo el banco de programas de prueba ha sido endurecido con las tres técnicas de endurecimiento software diseñadas (TMR1, TMR2 y SWIFT-R). Se ha realizado la verificación funcional de todos los programas endurecidos. Como era de esperar, el 100 % de los programas endurecidos han sido verificados y sus resultados coinciden con los resultados esperados. Es importante aclarar

```

...
load s0, sA
load s1, sB
return
;Output [00]:01,02,03,04,05

```

frestrepo@portatil-frestrepo: ~/doctorado/2_anno/repositorio_picohard/pro

Archivo Editar Ver Terminal Sólopas Ayuda

Sim_picohard 0.2 (alpha) Release [090710] (c) UniCAD - University of Alicante

>>> Simulation file: '../../../../../rtests hardening/01_bubbleSort.asm'

Check succeeded - Instructions simulated: 228

Instructions in original code: 46

Single simulation result: PASSED

Post-simulation ScratchPadRAM:

[00]:01	[01]:02	[02]:03	[03]:04	[04]:05	[05]:00	[06]:00	[07]:00
[08]:00	[09]:00	[0A]:00	[0B]:00	[0C]:00	[0D]:00	[0E]:00	[0F]:00
[10]:00	[11]:00	[12]:00	[13]:00	[14]:00	[15]:00	[16]:00	[17]:00
[18]:00	[19]:00	[1A]:00	[1B]:00	[1C]:00	[1D]:00	[1E]:00	[1F]:00
[20]:00	[21]:00	[22]:00	[23]:00	[24]:00	[25]:00	[26]:00	[27]:00
[28]:00	[29]:00	[2A]:00	[2B]:00	[2C]:00	[2D]:00	[2E]:00	[2F]:00
[30]:00	[31]:00	[32]:00	[33]:00	[34]:00	[35]:00	[36]:00	[37]:00
[38]:00	[39]:00	[3A]:00	[3B]:00	[3C]:00	[3D]:00	[3E]:00	[3F]:00

Post-simulation registers:

s0=4, s1=0, s2=5, s3=0, s4=0, s5=0, s6=4, s7=0, s8=4, s9=0, sA=3, sB=4, sC=0,
sD=0, sE=0, sF=0,

Figura 5.10: Verificación funcional exitosa de un programa

que este resultado era el único resultado aceptable, ya que si se hubiera obtenido otro resultado, sería indicador que algo ha fallado en el *endurecedor* o que las transformaciones realizadas por las técnicas de endurecimiento modifican el comportamiento de los programas y sus resultados finales, lo cual no es aceptable en ningún caso.

Impacto en el tamaño del código y el rendimiento de los programas (*overheads*)

Después de realizar el endurecimiento de los programas integrantes del banco de pruebas, se ha analizado el *overhead* del código fuente que ha provocado cada una de las técnicas de endurecimiento (medido en número de instrucciones). La Figura 5.13 presenta los resultados obtenidos. Estos resultados están normalizados con respecto a una línea base construida con el tamaño de código fuente de las versiones originales (no endurecidas) de los programas.

Se puede observar que las técnicas *TMR1* y *TMR2* presentan casi el mismo impacto sobre el tamaño del código. La media geométrica (calculada entre todos los resultados de los programas de prueba) del incremento en el tamaño del código es de $\times 2,59$ y $\times 2,65$ para *TMR1* y *TMR2* respectivamente; mientras que *SWIFT-R* causa un *overhead* mayor ($\times 3,08$). Este hecho se debe a que, tanto *TMR1* como *TMR2*, han sido implementadas sólo para proteger las instruccio-

```

...
load s0, sA
load s1, sB
return
[Output [00]:01,02,03,0F,05]

-----
Sim_picohard 0.2 (alpha) Release [090710] (C) UniCAD - University of Alicante
-----

>>> Simulation file: '../..../01_bubbleSort.asm'
Check failed - 1 error(s):
[1] ScratchPadRAM[3] = '4' does not match to expected value = 'f'
Instructions in original code: 46
Single simulation result: NOT PASSED
Post-simulation ScratchPadRAM:
[00]:01 [01]:02 [02]:03 [03]:04 [04]:05 [05]:00 [06]:00 [07]:00
[08]:00 [09]:00 [0A]:00 [0B]:00 [0C]:00 [0D]:00 [0E]:00 [0F]:00
[10]:00 [11]:00 [12]:00 [13]:00 [14]:00 [15]:00 [16]:00 [17]:00
[18]:00 [19]:00 [1A]:00 [1B]:00 [1C]:00 [1D]:00 [1E]:00 [1F]:00
[20]:00 [21]:00 [22]:00 [23]:00 [24]:00 [25]:00 [26]:00 [27]:00
[28]:00 [29]:00 [2A]:00 [2B]:00 [2C]:00 [2D]:00 [2E]:00 [2F]:00
[30]:00 [31]:00 [32]:00 [33]:00 [34]:00 [35]:00 [36]:00 [37]:00
[38]:00 [39]:00 [3A]:00 [3B]:00 [3C]:00 [3D]:00 [3E]:00 [3F]:00

Post-simulation registers:
s0=4, s1=0, s2=5, s3=0, s4=0, s5=0, s6=4, s7=0, s8=4, s9=0, sA=3, sB=4, sC=0,
sD=0, sE=0, sF=0,

```

Figura 5.11: Verificación funcional fallida de un programa

nes aritméticas y lógicas, y por el contrario, *SWIFT-R* se ha implementado de forma completa (para proteger todo el banco de registros), y por consiguiente, requiere la inserción de un número mayor de instrucciones para garantizar el endurecimiento. Aún así, nótese que para algunos programas del *benchmark*, esta diferencia no es significativa, como por ejemplo en el caso de: *aes* y *pid*; o incluso existen otros casos en que el *overhead* de código de *SWIFT-R* es ligeramente menor que el de las otras dos técnicas (*bub* y *pow*).

Asimismo, se ha comprobado el efecto que tiene el endurecimiento sobre el tiempo de ejecución de los programas, y por consiguiente, la degradación del rendimiento que produce la redundancia software. Los resultados obtenidos, también normalizados con respecto al tiempo de ejecución de las versiones no endurecidas, se presentan en la Figura 5.14.

Nótese una vez más que *TMR1* y *TMR2* provocan un *overhead* del tiempo de ejecución similar, siendo su media geométrica de $\times 2,58$ y $\times 2,41$ respectivamente; y *SWIFT-R* causa una degradación del rendimiento ligeramente superior de $\times 2,83$. El hecho que *SWIFT-R* cause un *overhead* mayor se debe, igual que antes, a que esta técnica ha sido aplicada completamente para proteger el banco de registros completo del microprocesador, mientras que *TMR1* y *TMR2* sólo se centran en dos tipos de instrucciones particulares (aritméticas y lógicas). Las principales diferencias se pueden observar en los programas en los cuales se utilizan principalmente otro tipo de instrucciones, además de las aritméticas y lógicas, como por ejemplo: desplazamientos, rotaciones e instrucciones

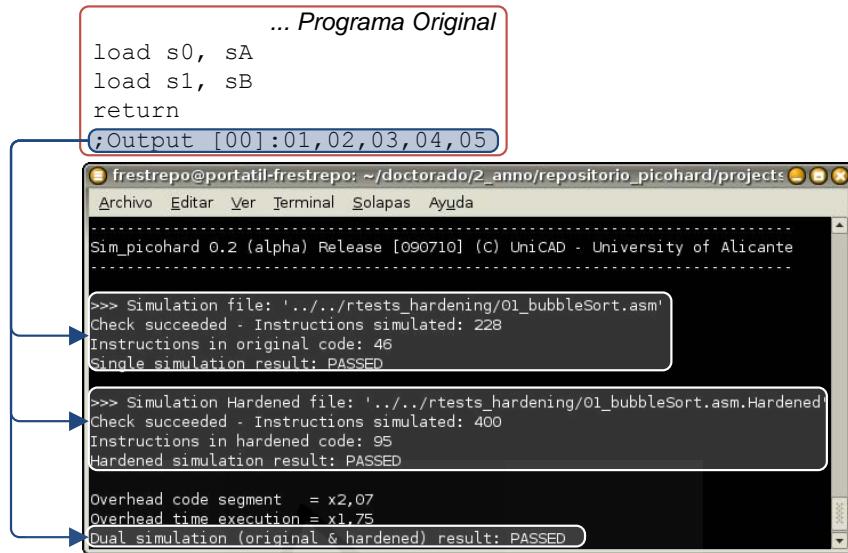


Figura 5.12: Verificación funcional de los programas endurecidos

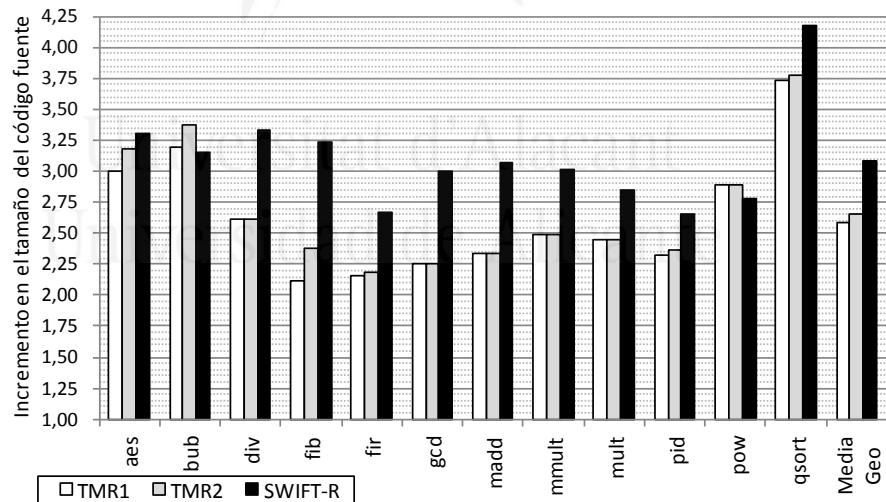


Figura 5.13: Impacto del endurecimiento sobre el tamaño del código fuente de los programas (normalizado)

de control de flujo (*aes, fir, pid, qsort*).

El incremento en el tamaño del código puede ser considerado como un problema considerable, ya que es posible que el código endurecido exceda el ta-

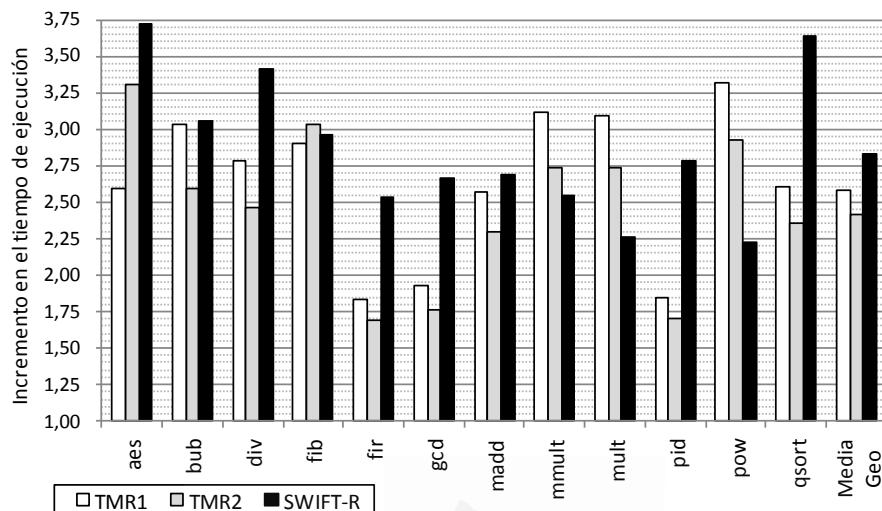


Figura 5.14: Impacto del endurecimiento sobre el tiempo de ejecución de los programas (normalizado)

maño total de la memoria de programa disponible; este hecho cobra más importancia en algunas aplicaciones de los sistemas embebidos donde el tamaño de la memoria es en ocasiones muy limitado (como en el caso particular de *PicoBlaze*). No obstante, desde el punto de vista de la confiabilidad, el incremento en el tiempo de ejecución es un problema aún más grave, ya que mientras mayor sea la duración del programa, mayor tiempo estará el sistema expuesto a un fallo ocasionado por un evento externo, como en el caso de los fallos transitorios inducidos por radiación. Este hecho debe ser considerado al evaluar la confiabilidad del sistema.

Evaluación de la confiabilidad

Para evaluar la calidad de las técnicas diseñadas, se ha dotado al *ISS* de un módulo capaz de simular fallos tipo *SEU* durante el proceso de simulación del programa. Este módulo de inyección permite inyectar aleatoriamente un *bit-flip*², simulando un fallo tipo *SEU* y afectando los datos almacenados en los registros del microprocesador (banco de registros, contador de programa, puntero de pila, banderas de la ALU, ...). Según el modelo de fallos *bit-flip*, se inyecta exactamente 1 fallo en cada ejecución del programa.

El procedimiento para simular un fallo con el *ISS* es el siguiente:

1. Se simula primero el programa completo sin inyectar fallos con el fin de conocer cuales son los resultados esperados y establecer cuál es el número total de instrucciones simuladas durante su operación normal.

²Cambio del estado lógico de un *bit* de 0 a 1 o de 1 a 0.

2. Se selecciona de forma aleatoria (mediante una distribución normal) el bit interno del microprocesador que será afectado por el *bit-flip*. Este bit se selecciona de entre todos los bits del banco de registros y los *flags* que expone el ISA del microprocesador original. Adicionalmente, también se han incorporado opciones para seleccionar de forma exacta el registro y el bit que será objeto del fallo. Esta funcionalidad permite hacer evaluaciones exhaustivas de la sensibilidad de ciertos registros a los SEUs.
3. Se selecciona de forma aleatoria (mediante una distribución normal) el número de la instrucción (dinámica) en la cual será simulado el fallo. En el *ISS* las unidades de tiempo son las instrucciones y no los ciclos de reloj. También, al igual que en el caso anterior, existen opciones especiales para depuración que permiten especificar el número exacto de la instrucción en la cual será simulado el fallo.
4. Se inicia una vez más la simulación del programa, pero esta vez, será inyectado un *bit-flip* en el bit elegido justo después de la simulación de la instrucción seleccionada.
5. Después de inyectar el *bit-flip* se continúa con la simulación del programa normalmente.
6. Por último, se evalúan los resultados obtenidos y se determina el efecto que ha tenido el fallo sobre el comportamiento del programa (verificación funcional).

La opción implementada en el ISS para llevar a cabo una campaña de inyección de fallos es (**--SEU [=VALUE]**). Esta opción permite ejecutar el procedimiento de inyección descrito anteriormente para un número **VALUE** de simulaciones, inyectando exactamente un *bit-flip* en cada ejecución. Esta opción es utilizada para evaluar el nivel de cobertura frente a fallos que proporcionan las técnicas de endurecimiento.

Como se mencionó en la Sección 2.6.2, los efectos ocasionados en el programa a causa de un fallo se tipifican de acuerdo a la clasificación dada al bit que haya sido objeto del fallo [Mukherjee et al., 2003], así:

- *unnecessary for Architecturally Correct Execution* (unACE): cuando el programa completa su ejecución y a pesar del fallo inyectado logra obtener los resultados esperados. Esta situación puede darse porque el fallo permaneció pasivo en el sistema hasta que finalizó la ejecución y no llegó a convertirse en un error; o bien, porque la técnica de tolerancia a fallos logró recuperar el error causado por el fallo (en el caso de los programas endurecidos).
 - *Silent Data Corruption* (SDC): cuando el programa termina su ejecución normalmente pero no obtiene los resultados esperados a causa del fallo inyectado, es decir, la ejecución del programa ha sido funcionalmente diferente a la operación normal del programa.
-

- *Hang o Bloqueado*: cuando el programa termina su ejecución de forma anormal o permanece iterando en un ciclo infinito a causa del fallo inyectado.

Nótese que SDC y *Hang* son efectos indeseables para el sistema y se pueden clasificar en su conjunto como fallos que han ocurrido en bits necesarios para la correcta ejecución de la arquitectura o ACE.

Además de la capacidad para simular fallos, el ISS proporciona información sobre la composición interna de los programas simulados, para saber qué porcentajes del número total de instrucciones simuladas corresponden a cada tipo de instrucción (directivas, control de flujo, interrupción, aritméticas, lógicas, desplazamientos, rotaciones, almacenamientos, entrada/salida). Esta información es útil para establecer la correlación existente entre estos datos y los resultados obtenidos por las técnicas de endurecimiento, por ejemplo, cuando los programas están compuestos por un mayor o menor porcentaje de algún tipo de instrucción y deben ser protegidos de formas diferentes.

De igual forma, para evaluar la confiabilidad del sistema, en la Sección 2.6.2 se presentó la métrica MWTF [Reis et al., 2005a], que es una generalización de la métrica conocida como *tiempo medio hasta la avería* o MTTF. La métrica MWTF fue escogida para este caso de estudio porque captura la solución de compromiso que hay entre el nivel de fiabilidad (porcentaje de fallos unACE) y el tiempo de ejecución del sistema.

Se han realizado dos experimentos para evaluar la confiabilidad que ofrece cada una de las técnicas de endurecimiento diseñadas:

- En primer lugar, se ha utilizado el ISS para hallar las estimaciones preliminares de confiabilidad.
- Posteriormente, se ha utilizado el *FT-Unshades* para validar las estimaciones de confiabilidad obtenidas en el primer experimento y obtener valores en condiciones realistas de ejecución.

En el primer experimento, para cada uno de los doce programas que conforman el banco de pruebas (versiones endurecidas con las tres técnicas y versión no-endurecida), se han realizado 5000 simulaciones de su ejecución para obtener datos estadísticamente significativos. De acuerdo al modelo de fallo *bit-flip*, en cada una de las simulaciones se ha inyectado un único fallo durante la ejecución del programa. El fallo se ha simulado mediante un *bit-flip* en un bit seleccionado de forma aleatoria entre todos los bits del banco de registros de *PicoBlaze* (16 registros de 8 bits cada uno).

La Figura 5.15 muestra los resultados obtenidos por el ISS. Por un lado, presenta los porcentajes de la clasificación de los fallos inyectados de acuerdo a su efecto en el comportamiento del programa, organizados por columnas apiladas donde cada una de las columnas representa una de las versiones de los programas y contiene la distribución obtenida de unACE, SDC y *Hang*. Además en la misma Figura se muestra, en un eje secundario (en escala logarítmica), el

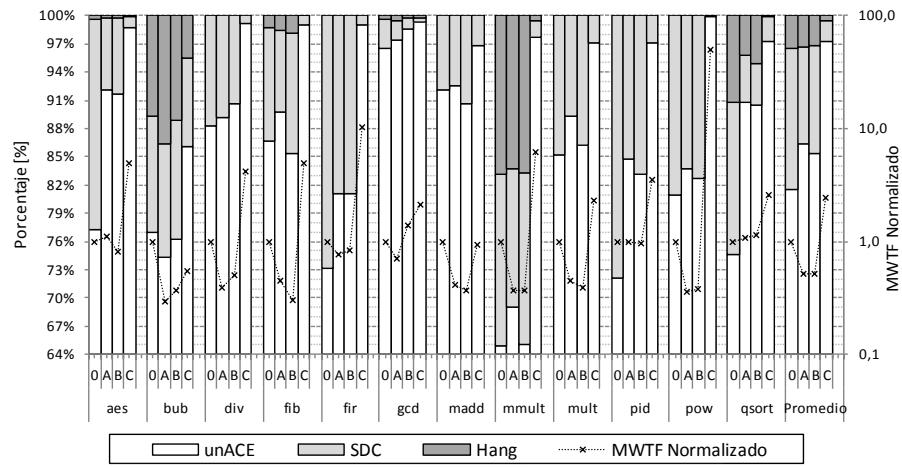


Figura 5.15: Porcentajes de clasificación de fallos y MWTF normalizado para las versiones: no-endurecida (0), TMR1 (A), TMR2 (B) y SWIFT-R (C) - Campaña de prueba realizada en el ISS contra el banco de registros del microprocesador

MWTF obtenido para cada programa. Los resultados de MWTF están normalizados con respecto a la versión no-endurecida de cada programa.

La cobertura frente a fallos que ofrece cada una de las técnicas está determinada por el porcentaje de fallos clasificados como unACE. El promedio de estos porcentajes es de un: 81.50 % para las versiones no-endurecidas, 86.32 % para TMR1, 85.34 % para TMR2 y 97.31 % para SWIFT-R. Estos valores representan el porcentaje de los fallos inyectados que no ha provocado ningún efecto indeseado en el comportamiento del programa durante su ejecución (sin tener en cuenta la micro-arquitectura del microprocesador, ya que el ISS no tiene acceso a ésta). A pesar de que estos resultados pueden ser considerados tan sólo estimaciones, también constituyen información valiosa, ya que permiten comparar los resultados obtenidos entre sí, y de esta forma, comparar las bondades de las diferentes técnicas diseñadas. Más adelante en esta misma sección, sin embargo, se demostrará que estas estimaciones se aproximan a la realidad.

Para tomar decisiones acerca del diseño del sistema, se deben tener en cuenta no sólo estos resultados de cobertura frente a fallos, sino también los *overheads* de tamaño de código y tiempo de ejecución. En este caso, como se observa en la Figura 5.15, las técnicas TMR1 y TMR2 no ofrecen una mejora significativa de la cobertura frente a fallos. SWIFT-R, por otro lado, aunque presenta una mejora importante de la cobertura frente a fallos, también causa los mayores incrementos en tiempo de ejecución y tamaño de código; factores éstos que deben ser considerados dependiendo de las restricciones de la aplicación que se esté diseñando. Este tipo de análisis puede motivar importantes decisiones de diseño acerca del máximo nivel de protección que se puede aplicar mediante técnicas software a una aplicación específica, para posteriormente complemen-

tar la protección mediante técnicas basadas en hardware. Por lo tanto, la explotación inicial del espacio de diseño del endurecimiento software constituye una estrategia muy valiosa para obtener una partición hardware/software óptima del sistema, de acuerdo con los principios del co-diseño hardware/software.

Nótese en la Figura 5.15 que no sólo *SWIFT-R* ofrece una mejor cobertura frente a fallos comparado con *TMR1* y *TMR2*, sino que también alcanza un valor de MWTF normalizado $\times 2.42$ veces mayor que la versión sin endurecer (en promedio), mientras que el MWTF normalizado de *TMR1* y *TMR2* únicamente llega a $\times 0.52$ en promedio (el mismo valor para ambas). Este último valor (por debajo de la línea base $\times 1.00$) significa que los programas endurecidos con *TMR1* y *TMR2* son más propensos a errores que sus versiones sin endurecer, lo cual confirma los análisis previos que se habían realizado durante la discusión preliminar.

Para validar las estimaciones obtenidas durante el primer experimento por el ISS, se ha realizado un segundo experimento utilizando *FT-Unshades*, en el cual se ha realizado la evaluación de la confiabilidad utilizando las versiones reales del sistema implementadas usando el microprocesador *RTL PicoBlaze*. Para este segundo experimento, se ha diseñado una campaña de inyección de fallos por emulación, con características similares a las de la campaña diseñada para el ISS: 5000 ejecuciones de cada programa (versiones endurecidas y no-endurecidas), emulando un único fallo tipo SEU durante cada ejecución. Cada fallo ha sido emulado en un ciclo de reloj seleccionado de forma aleatoria de toda la duración de la ejecución.

Antes de diseñar la campaña de inyección de fallos, se realizaron un conjunto de simulaciones para estimar el *tiempo crítico de recuperación*, en el cual se abarcaban los diferentes esquemas de recuperación. Esta tarea se realizó para evitar una dependencia indeseable del valor de este parámetro en la clasificación incorrecta de los fallos. Se encontró que un tiempo de recuperación de 1023 ciclos de reloj era adecuado para todos los programas de prueba y para las tres técnicas de endurecimiento de este caso de estudio.

Los resultados obtenidos en el segundo experimento han sido clasificados de la misma forma que para la evaluación realizada con el ISS. La Figura 5.16 presenta los resultados obtenidos.

En este caso, los porcentajes de fallos unACE que proporciona cada método son los siguientes: 87.32 % para la versión no-endurecida, 90.77 % para *TMR1*, 90.08 % para *TMR2* y 98.10 % para *SWIFT-R*. Como se puede observar, estos resultados confirman los resultados preliminares obtenidos durante la evaluación de la confiabilidad realizada mediante simulación con el ISS. Aunque los porcentajes obtenidos con *FT-Unshades* son ligeramente mayores que los obtenidos con el ISS, mantienen las mismas tendencias. Asimismo, este hecho puede observarse también al analizar los resultados del MWTF normalizado. Por lo tanto, el ISS sirve como un buen predictor de la confiabilidad cuando se comparan diferentes técnicas software de endurecimiento.

Debe tenerse en cuenta que los altos porcentajes de cobertura frente a fallos obtenidos en los dos experimentos por las versiones no endurecidas se deben a que la prueba de inyección de fallos fue realizada sobre todo el banco de

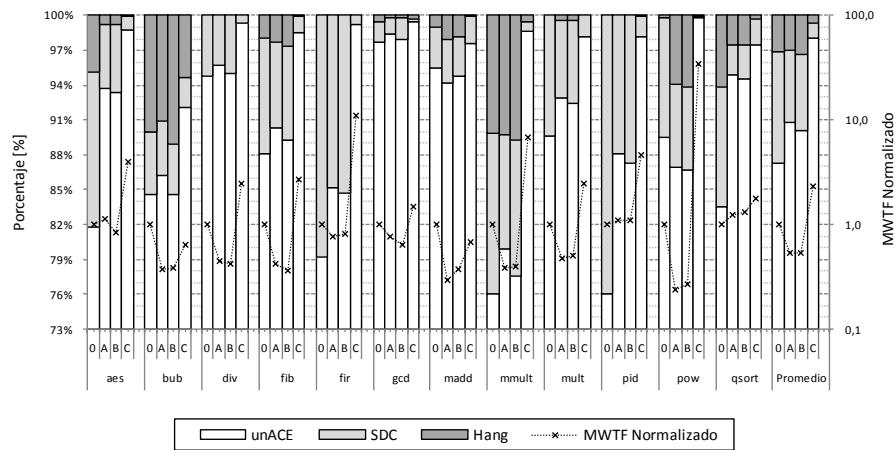


Figura 5.16: Porcentajes de clasificación de fallos y MWTF normalizado para las versiones: no-endurecida (0), TMR1 (A), TMR2 (B) y SWIFT-R (C) - Campaña de prueba realizada en FT-Unshades contra el banco de registros del microprocesador

registros del microprocesador, aunque los programas no usaran la totalidad de los 16 registros disponibles en *RTL PicoBlaze*. Por lo tanto, al inyectar un fallo en un bit de un registro no utilizado, éste es considerado como unACE porque no afecta el comportamiento del sistema de ninguna manera. Se ha decidido incluir también estos datos, por un lado, para mantener la homogeneidad de los experimentos con respecto a los realizados por otros investigadores [Oh et al., 2002c, Oh et al., 2002b, Reis et al., 2005b, Reis et al., 2007, Rebaudengo et al., 2001], y por otro lado, debido a que esta situación es la más realista.

Como conclusión general de esta sección, se puede decir que a partir de las técnicas diseñadas y los experimentos realizados, se ha demostrado lo siguiente: en primer lugar, la flexibilidad que ofrece SHE para el diseño y evaluación de técnicas de endurecimiento software; y en segundo lugar, la relevancia de la información que proporciona a la hora de guiar la exploración del espacio de co-diseño.

5.3. Desarrollo de estrategias híbridas de endurecimiento

Con el objetivo de ilustrar la aplicabilidad de la propuesta de *co-endurecimiento hardware/software*, se ha desarrollado una técnica híbrida de endurecimiento a medida cuya principal característica es la flexibilidad, ya que puede ser aplicada de forma selectiva tanto en hardware como en software.

Para esto, el espacio de diseño de las técnicas de endurecimiento se reduce a las técnicas: SWIFT-R en el software, y TMR en el hardware. Sin embargo, es importante resaltar el hecho de que en este caso, ambos enfoques se pueden aplicar al sistema de forma selectiva, lo cual permite la exploración detallada (de grano fino) del espacio de diseño y facilita el desarrollo de sistemas embedidos que cumplan de forma óptima con todas los requisitos exigidos.

Protección software: SWIFT-R selectivo

La flexibilidad de SHE permite la exploración detallada de la aplicación de las técnicas software de forma selectiva, es decir, proteger solamente partes específicas de los programas o de los recursos de la arquitectura. SWIFT-R ha sido seleccionada porque fue la estrategia de endurecimiento software que evidenció mejores resultados de confiabilidad en la evaluación presentada en la sección anterior.

SWIFT-R ha sido implementado de forma tal que es posible realizar su aplicación únicamente a diferentes recursos de la arquitectura. En concreto, es posible seleccionar diferentes subconjuntos de registros para ser protegidos de entre todos los registros presentes en el banco de registros del microprocesador. Esta estrategia busca reducir el impacto del endurecimiento software sobre el tamaño del código fuente y la degradación del rendimiento de los programas, y al mismo tiempo, mantener un nivel aceptable de cobertura frente a fallos.

En el caso del microprocesador *PicoBlaze*, es posible tener versiones endurecidas de los programas donde sólo algunos de los 16 registros internos que posee estén endurecidos por software. Por ejemplo, una versión del software en la cual se aplique SWIFT-R tan sólo a los registros $\{0 - 2 - F\}$, significa que únicamente dichos registros estarán endurecidos (numerados en notación hexadecimal).

La novedosa implementación selectiva de SWIFT-R es posible gracias a la flexibilidad que ofrece a SHE la *esfera de replicación* (SoR). El funcionamiento básico de SWIFT-R selectivo consiste en trasladar los recursos que no se desean endurecer afuera de la SoR. En el caso del ejemplo anterior, se dejan dentro de la SoR únicamente los registros $\{0 - 2 - F\}$ y los demás se ubican fuera de la esfera, en la zona sin protección.

Con el objeto de ilustrar este nuevo algoritmo, la Figura 5.17 presenta un ejemplo básico de un fragmento de programa endurecido mediante SWIFT-R selectivo aplicado a distintos conjuntos de registros. En las diferentes columnas de la Figura se pueden apreciar las diferencias cuando se seleccionan distintos subconjuntos de registros para ser protegidos.

Nótese que es necesario la inserción de votadores para verificar la consistencia de los datos cuando una instrucción cualquiera provoca un flujo de datos desde dentro hacia fuera de la SoR. Esto se puede apreciar claramente en los ejemplos de la Figura 5.17, por ejemplo, un caso que llama la atención se puede ver en el fragmento “*Reg 1 protegido*” cuando se inserta un votador por mayoría para *s1* justo antes de la instrucción *ADD s0, s1*. En este ejemplo solamente el registro *s1* se considera dentro de la SoR. Por lo tanto, al ejecutarse la ins-

No endurecido	Reg "0" protegido	Reg "1" protegido	Regs "0-1" protegidos
LOAD s0, 00	LOAD s0, 00 <i>Copias para s0</i>	LOAD s0, 00	LOAD s0, 00 <i>Copias para s0</i>
LOAD s1, 2A	LOAD s1, 2A <i>Copias para s1</i>	LOAD s1, 2A <i>Votador para s1</i>	LOAD s1, 2A <i>Copias para s1</i>
ADD s0, s1	ADD s0, s1 ADD s0', s1 ADD s0'', s1 <i>Votador para s0</i>	ADD s0, s1 <i>Votador para s1</i>	ADD s0, s1 ADD s0', s1' ADD s0'', s1'' <i>Votador para s0</i> <i>Votador para s1</i>
STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)

Figura 5.17: Ejemplos de SWIFT-R selectivo

trucción ADD s0, s1 (cuya función es hacer $s0 = s0 + s1$) habrá un flujo de datos desde s1 a s0, o en otras palabras, habrá un flujo de datos desde el interior de la SoR hacia el exterior, y por consiguiente, se debe insertar un votador que verifique que el valor almacenado en s1 es correcto, antes que éste abandone la esfera.

El número máximo de versiones endurecidas de los programas depende del número de registros que se utilicen en el programa. Por ejemplo, para un programa que sólo utilice en su código dos registros del banco de registros (como en el caso del ejemplo de la Figura anterior), solamente existirán cuatro (4) versiones del software distintas: la versión no-endurecida (original), la versión con el primer registro endurecido, la versión con el segundo registro endurecido, y la versión con ambos registros endurecidos. Por lo tanto, el número máximo de versiones endurecidas es igual al número de posibles combinaciones (sin repetición) entre los registros utilizados en el programa. Es decir, el número máximo de versiones es igual al número combinatorio $\binom{m}{n}$, donde $m > n$, y m representa el número total de registros del banco y n el subconjunto escogido.

No obstante, es importante tener en cuenta el número de registros que se necesitan para poder aplicar las técnicas de endurecimiento software y crear las copias redundantes de los registros utilizados. En el caso de SWIFT-R, son necesarias dos copias adicionales por cada registro utilizado en el programa original, es decir, en total son necesarios $3n$ registros para aplicar SWIFT-R, donde n es el número de registros utilizado en el programa original (n originales + $2n$ adicionales). Nótese que como el microprocesador *PicoBlaze* solamente tiene 16 registros disponibles, al aplicar SWIFT-R al total de los registros utilizados, el número máximo de registros que se deben utilizar en los programas originales es de 5, ya que una vez que se haya aplicado la técnica de endurecimiento, el programa completamente endurecido usará 15 registros (los 5 que usaba originalmente y 10 copias redundantes).

En el caso del endurecimiento de un programa que utilice más de 5 registros en su versión original, se debe usar SWIFT-R selectivo para proteger el programa, priorizando un máximo de 5 registros para ser endurecidos y dejando los demás registros sin protección (por lo menos en el software).

Protección hardware: TMR selectivo

Con respecto al hardware, la estrategia de tolerancia a fallos que se ha diseñado consiste en el endurecimiento selectivo e incremental de varios recursos internos de la arquitectura del microprocesador. Fue posible llevar a cabo esta estrategia ya que se está trabajando con un microprocesador *soft core* (*PicoBlaze*) y se puede modificar su estructura interna con facilidad al estar descrito en *VHDL*. No obstante, hay que tener en cuenta que si se estuviera trabajando con un microprocesador COTS, esto no sería posible y deberían buscarse otras alternativas, como por ejemplo: proteger los puertos de entrada/salida del microprocesador mediante estructuras redundantes externas, utilizar enfoques de redundancia modular a nivel de sistema, entre otros.

En este caso de estudio, la estrategia de endurecimiento hardware se fundamenta en la aplicación de la conocida técnica TMR de forma selectiva a diferentes recursos de la arquitectura. La estrategia de protección hardware busca además, complementar la protección ofrecida por la estrategia de protección del software. Por lo tanto, los recursos que se protegen mediante hardware son aquellos a los que no es posible aplicar protección mediante software usando SWIFT-R, como por ejemplo: el contador del programa, las banderas (*flags*) de la ALU, el puntero de pila, y aquellos registros en el *pipeline* del microprocesador. De esta forma, se han diseñado cinco (5) versiones distintas del microprocesador:

- **P0:** RTL *PicoBlaze* sin endurecer (versión original).
- **P1:** microprocesador con redundancia hardware tipo TMR para el contador de programa — *PC* (10 bits), los *flags* de la ALU — *zero* y *carry* (2 bits), y el puntero de pila — *SP* (5 bits).
- **P2:** redundancia hardware en todos los registros internos del *pipeline* del microprocesador (52 bits). Estos registros no son accesibles desde el ISA.
- **P3:** microprocesador con redundancia hardware para el *PC*, los *flags* de la ALU, el *SP*, y todos los registros del *pipeline*. Esta versión equivale a *P1 + P2*.
- **P4:** versión del microprocesador completamente protegida mediante TMR en todos los registros, es decir, redundancia hardware para el *PC*, los *flags* de la ALU, el *SP*, todos los registros del *pipeline*, y el banco de registros (16 registros de 8 bits cada uno, en total 128 bits). Esta versión equivale a *P3 + banco de registros protegido*.

Pese a que los enfoques de redundancia hardware suelen ser muy efectivos para la mitigación de fallos, hay que tener en cuenta que la redundancia hardware se traduce en incrementos en los costes del hardware, lo cual para muchos diseños puede ser una limitación importante. En este caso de estudio, se han considerado los resultados reportados por la herramienta de síntesis

(Xilinx XST v10.1) durante la implementación de los sistemas como una estimación de los costes hardware. Estos resultados se expresan en términos de: lógica secuencial (*flip-flops* y *latches*), lógica combinacional, y memoria RAM (en bloques y distribuida).

La Figura 5.18 presenta los costes hardware de cada una de las versiones endurecidas del microprocesador. Estos costes están normalizados con respecto a la versión *RTL PicoBlaze* no endurecida (*P0*). Como ninguna de las versiones endurecidas utiliza una mayor cantidad de RAM con respecto a la versión original, los costes de RAM no se representan en la Figura.

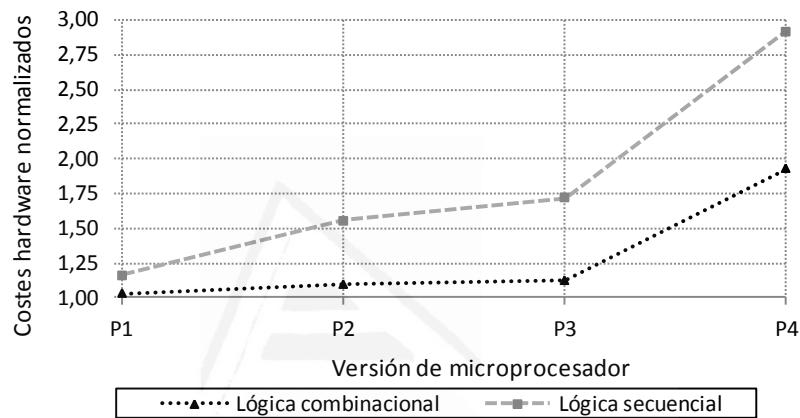


Figura 5.18: Costes hardware para cada versión del microprocesador normalizados con respecto a *P0*

Puede observarse que los costes hardware se incrementan ligeramente para el microprocesador *P1* (con respecto a *P0*) porque en esta versión se protegen tan sólo unos pocos registros (*PC*, *flags* y *SP*); mientras que cuando se protegen los registros del *pipeline* (mucho más numerosos), los costes hardware se elevan considerablemente (*P2*, *P3* y *P4*). También, nótense el gran incremento en los costes hardware que se produce al endurecer el banco de registros (*P4*), lo cual hace que la utilización de esta versión del microprocesador resulte inviable para muchas aplicaciones de los sistemas embebidos.

5.3.1. Casos de *co-endurecimiento hardware/software*

A continuación se presenta el proceso de *co-endurecimiento* de tres aplicaciones típicas de los sistemas embebidos con el fin de validar el enfoque de forma experimental. Las aplicaciones seleccionadas para ello son:

- Controlador *Proportional Integral Derivative* (PID).
- Filtro *Finite Impulse Response* (FIR).

- Multiplicación de matrices (*mmult*).

La estrategia de protección está basada, en todos los casos, en el endurecimiento selectivo de los sistemas a nivel de hardware y software: *SWIFT-R selectivo* en software; y *TMR selectivo* en hardware. No obstante, en cada una de los casos se presentan diferentes resultados de interés, ya que cada uno de los casos persigue los siguientes objetivos:

1. Controlador PID: por un lado, se busca demostrar la flexibilidad de la infraestructura presentada (SHE + *FT-Unshades*) para explorar el espacio de soluciones; y por otro lado, se hace énfasis en la discusión de los diferentes niveles de compromiso que se obtienen entre los distintos requisitos de diseño.
2. Filtro FIR: se verifica la utilidad que tienen las estadísticas que ofrece el ISS acerca de la aplicación con el fin de guiar el proceso de *co-endurecimiento*, y también se analizan los resultados de confiabilidad usando la métrica MWTF, lo cual facilita el análisis de compromisos.
3. Multiplicación de matrices (*mmult*): se busca presentar un ejemplo de *co-endurecimiento* guiado por la aplicación, donde de forma más realista no se implementan todas las versiones del sistema de forma exhaustiva, sino que se aprovecha el flujo de diseño presentado para restringir el área del espacio de soluciones a explorar.

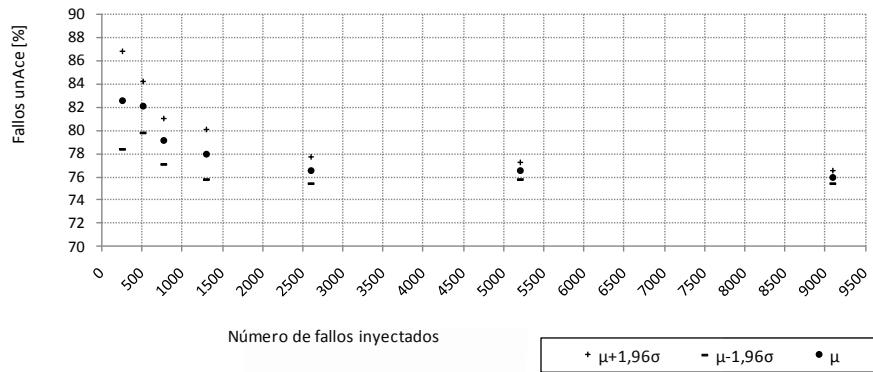
En los siguientes apartados se presentan los resultados obtenidos (y su discusión correspondiente) al explorar el espacio de *co-endurecimiento* de cada una de estas aplicaciones. Sin embargo, antes de pasar a discutir los resultados obtenidos se presenta el procedimiento realizado para calibrar *FT-Unshades*. Esto es necesario para realizar adecuadamente las campañas de inyección de fallos y evaluar la confiabilidad de cada una de los sistemas diseñados.

Calibración de la herramienta *FT-Unshades*

Antes de realizar la evaluación de la confiabilidad de cada una de los sistemas, se ha llevado a cabo un experimento para calibrar *FT-Unshades*. Con el objetivo de determinar el mínimo número de pruebas de inyección de fallos que se debía realizar, de forma tal que la cantidad de fallos inyectados fuera representativa y, de esta forma, encontrar resultados precisos de confiabilidad.

En este sentido, se han realizado varias campañas de inyección de fallos de forma incremental, aumentando gradualmente el número de fallos inyectado en cada ocasión. Las campañas incrementales se repitieron un total de 10 veces para obtener valores estadísticamente representativos. Cada campaña de inyección de pruebas realizaba ataques al banco de registros del microprocesador *P0*. La versión del programa seleccionada para este experimento fue la versión no-endurecida de la aplicación *mmult*, ya que constituye el escenario del peor de los casos, es decir, cuando los fallos afectaban más al sistema (según se puede observar en las figuras 5.15 y 5.16).

Los resultados obtenidos se presentan en la Figura 5.19. Se pudo evidenciar que el intervalo de confianza del 95 % es menor que $\pm 1.0\%$ después de 5000 - 5200 fallos inyectados (inyectando sólo un fallo por ejecución).



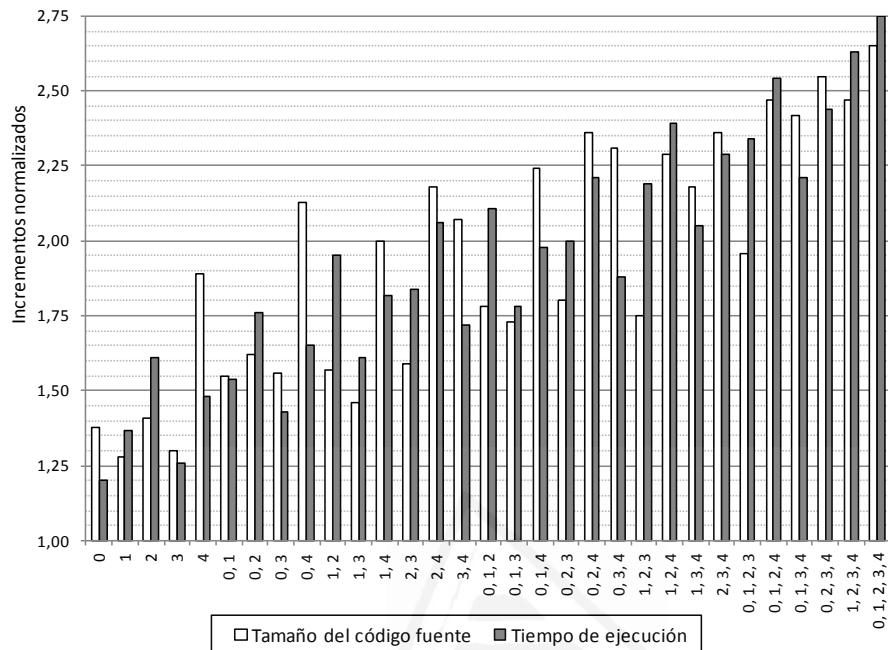


Figura 5.20: Impacto del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones de la aplicación PID (datos normalizados)

registro aporta a estos *overheads* de forma diferente. Por su parte el registro 4 incide de forma notoria sobre el tamaño del código (el *overhead* de código se incrementa hasta $\times 1.89$ solamente cuando se endurece este registro), mientras que el registro 2 afecta principalmente el tiempo de ejecución (el *overhead* de tiempo aumenta hasta $\times 1.61$ solamente cuando se protege este registro).

Con el fin de evaluar la confiabilidad de cada una de las versiones del sistema usando *FT-Unshades*, en esta ocasión se ha diseñado una campaña de inyección de fallos que ataca a todo el sistema, y no sólo al banco de registros (como ocurría en la campaña diseñada en la Sección 5.2.2). Cada campaña realiza ataques selectivos en diferentes subconjuntos de registros de la arquitectura del microprocesador, incluyendo: banco de registros, contador de programa - *PC*, *flags* de la *ALU*, puntero de pila - *SP*, y registros internos del *pipeline*. En cada uno de estos subconjuntos de registros se han emulado 5000 fallos SEU (uno por ejecución) en un ciclo de reloj seleccionado de forma aleatoria entre todos los ciclos de reloj que tarda la ejecución. De esta forma, se han inyectado un total de 25000 fallos en cada una de las versiones del sistema.

En este caso, se ha analizado inicialmente la confiabilidad de cada versión del software endurecido ejecutándose en la versión no-endurecida del micro-

procesador ($P0$). Para esto se ha llevado a cabo una campaña de inyección de fallos en *FT-Unshades* con las características descritas antes. La Figura 5.21 muestra la clasificación de los fallos inyectados durante las campañas de prueba para cada versión del sistema (solamente endurecidas por software). Estos porcentajes corresponden al promedio ponderado de los resultados obtenidos en los ataques selectivos a cada parte interna del microprocesador, asumiendo la misma probabilidad de fallo para todos los bits.

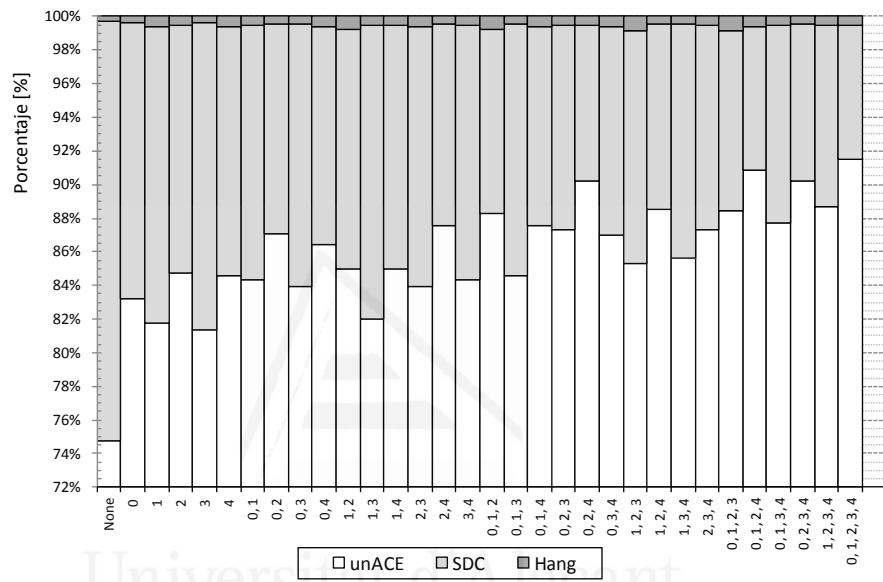


Figura 5.21: Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en software (microprocesador $P0$) para la aplicación *PID*

Nótese que utilizando únicamente el endurecimiento software es posible obtener una mejora considerable de la cobertura frente a fallos, hasta alcanzar un 91.46 % de fallos unACE en el programa completamente endurecido (versión con protección en todos los registros utilizados). Además, en comparación con la fiabilidad ofrecida por el programa no-endurecido (versión “None” en la Figura), la cual es de 74.72 % de fallos unACE, la fiabilidad de todas las versiones endurecidas resulta muy superior. Existen varias versiones con protección intermedia que pueden resultar adecuadas para muchas aplicaciones dependiendo de los requisitos. Por ejemplo, al aplicar la protección a los registros 2 y 4 se produce una mejora importante de la fiabilidad del sistema, ya que la protección de cada uno de estos dos registros críticos contribuye considerablemente a mejorar la confiabilidad del sistema. En este mismo sentido, un resultado interesante corresponde a la versión del sistema con protección en el subconjunto de registros $\{0, 2, 4\}$, donde se alcanza un porcentaje de fallos unACE de 90.23 %, el cual se aproxima al de la versión con protección en todos

los registros.

Analizando los resultados obtenidos hasta el momento, el diseñador puede explorar el espacio de diseño por parte del software, mediante la representación de distintos niveles de compromiso entre el tamaño del código, el rendimiento y la cobertura frente a fallos, y de esta forma, seleccionar cuales son las mejores versiones del software para ser complementadas mediante enfoques de protección hardware y seguir adelante con los análisis. Sin embargo, en este caso, con el objetivo de ilustrar aún más las posibilidades del *co-endurecimiento*, se han implementado todas las versiones hardware/software posibles. Cada sistema ha sido sintetizado e implementado usando la herramienta: *Xilinx ISE 10.1 suite*. Esto significa que se han configurado todas las versiones posibles del software endurecido en cada uno de las versiones del hardware endurecido. En el caso del PID, se tienen 32 versiones distintas del software ejecutándose en 4 versiones diferentes del hardware (P_0, P_1, P_2, P_3), y además, el programa no-endurecido ejecutándose en la versión del microprocesador completamente endurecida (P_4); en total, se han desarrollado 129 versiones diferentes.

De la misma manera que para las versiones del sistema con el hardware no endurecido, se ha evaluado la confiabilidad de cada una de las versiones del sistema mediante una campaña de inyección de fallos en *FT-Unshades* con las mismas características que antes. La Figura 5.22 ilustra los porcentajes de clasificación de fallos obtenidos para cada configuración hardware/software. En esta figura se presentan los resultados de cada versión del software ejecutándose en los microprocesadores P_1, P_2 , y P_3 . Los resultados del microprocesador P_0 ya han sido expuestos en la Figura 5.21, y los resultados del microprocesador P_4 tampoco se presentan ya que el 100% de los fallos inyectados fueron clasificados como unACE (como se esperaba).

Se puede observar que la confiabilidad aumenta considerablemente (hasta 97.13% de fallos unACE) al combinar *SWIFT-R* con protección hardware sólo en unos pocos registros críticos, tales como: *PC*, *flags* de la *ALU*, y *SP* (microprocesador P_1). Se puede apreciar también que los resultados obtenidos para la versión P_2 muestran que la redundancia hardware en los registros internos del *pipeline* apenas mejora levemente la confiabilidad del sistema (en el mejor de los casos alcanza un 93.11% de unACE), aunque la cantidad de registros protegidos en esta versión es mayor que en P_1 . No obstante, la protección hardware de ambos subconjuntos de registros es complementaria, y por esto, los mayores niveles de confiabilidad se obtienen en P_3 (hasta 98.78% unACE), que combina la protección de P_1 y P_2 .

Por otra parte, se deben considerar también los costes hardware de cada una de las versiones endurecidas parcialmente del microprocesador dentro de los análisis de compromisos que se hacen al explorar el espacio de diseño (véase la Figura 5.18).

A manera de ejemplo, la Figura 5.23 muestra la clase de análisis que el diseñador puede realizar. Este ejemplo presenta los resultados de una estrategia de endurecimiento incremental; se presentan los niveles de cobertura frente a fallos que alcanza cada una de las versiones preseleccionadas, y al mismo tiempo, se ilustran sus costes hardware para facilitar los análisis.

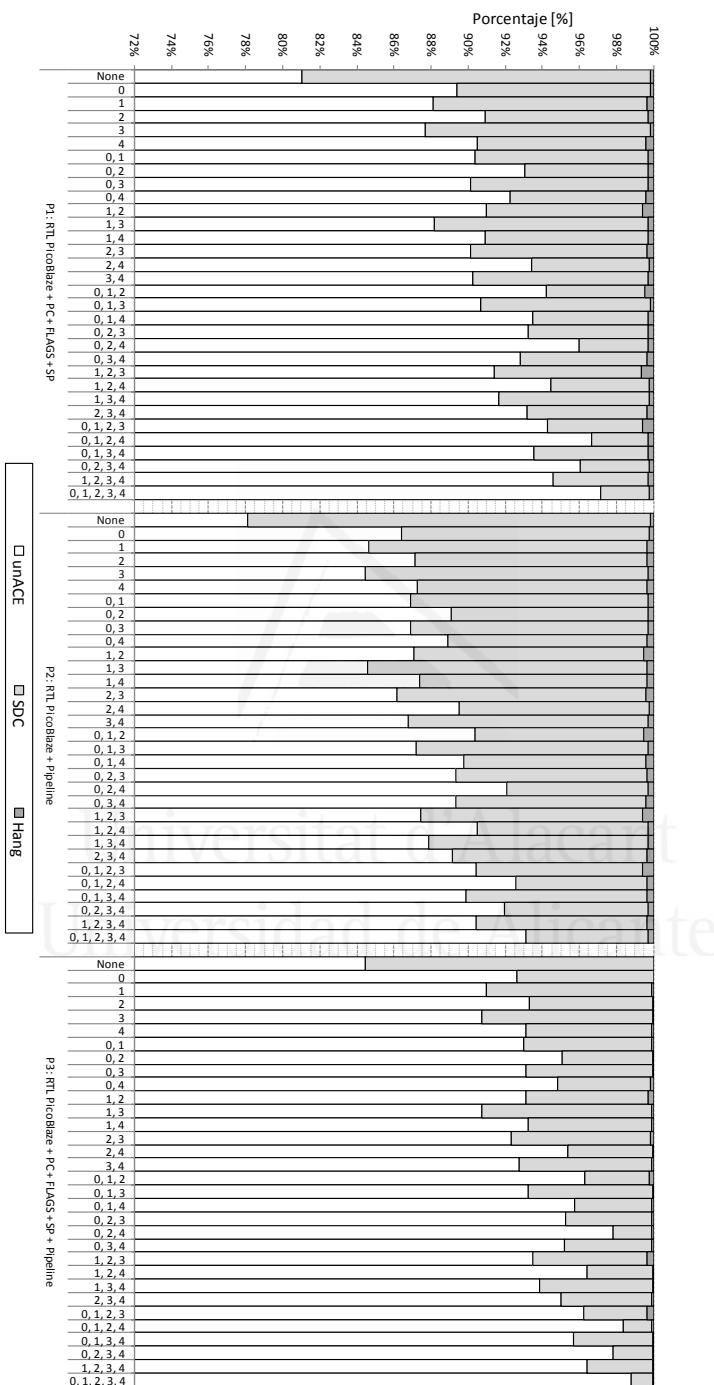


Figura 5.22: Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para la aplicación *PID*

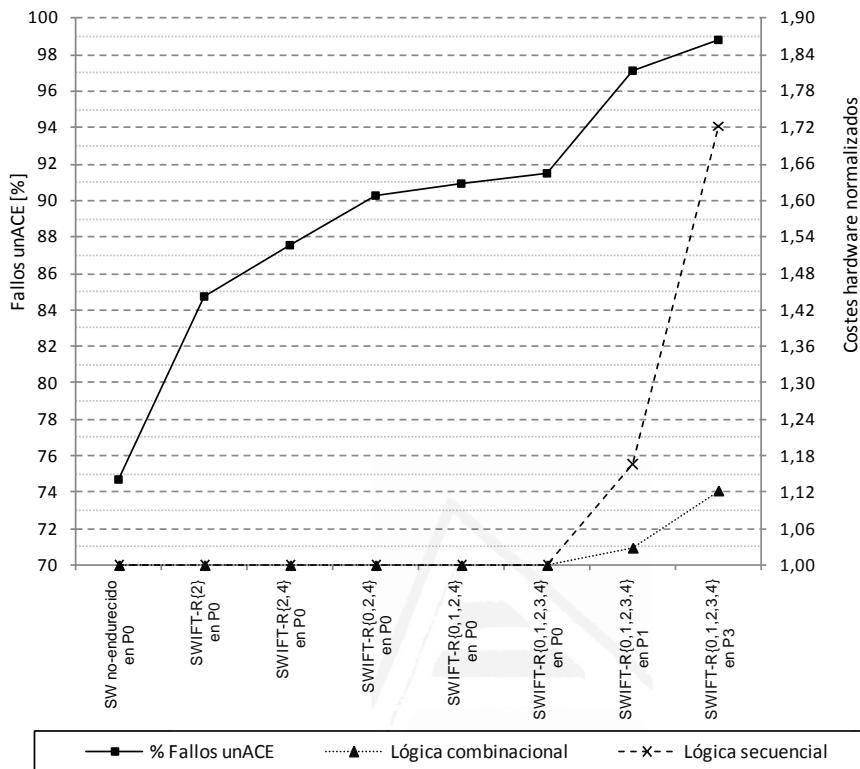


Figura 5.23: PID: Ejemplo de una estrategia de endurecimiento incremental presentando el porcentaje de fallos unACE y los costes hardware normalizados

Como se puede ver en la Figura 5.23, cuando se protege mediante software únicamente el registro 2 hay un incremento considerable en la cobertura frente a fallos, y posteriormente, mientras se va aplicando SWIFT-R de forma incremental al resto de los registros, la cobertura frente a fallos continúa mejorando ligeramente. Como hasta el momento no se utilizan las versiones protegidas del hardware, todavía no existe un coste hardware adicional en el sistema. No obstante, la segunda mejora importante en la cobertura frente a fallos puede observarse cuando se protegen algunos registros críticos en el hardware (microprocesador P_1), aunque los costes hardware también empiezan a crecer.

La figura anterior podría corresponder al camino de exploración, a seguir por el diseñador, donde sólo se analizan ciertos puntos del espacio de soluciones en los que se obtienen las mejores soluciones de compromiso. La forma de guiar al usuario por ese camino (como se mostrará en los siguientes casos) se realiza a través de la información proporcionada en las etapas previas del flujo de *co-endurecimiento*.

Estos resultados se encuentran publicados en [Cuenca-Asensi et al., 2011b].

Filtro FIR

En el caso de la aplicación del filtro FIR, para facilitar la labor del diseñador, se utiliza toda la información que proporciona SHE acerca del programa sin endurecer. Como en este caso, la técnica que se ha diseñado para el software (*SWIFT-R selectivo*) permite seleccionar diferentes subconjuntos de registros del microprocesador para ser protegidos, el informe de utilización de recursos del ISS permite identificar cuales son los recursos más críticos y ayuda a priorizar los registros que deben ser protegidos mediante software. En este informe se reúne información acerca de: el *número de accesos* a cada registro del banco de registros del microprocesador, es decir, el número de veces que cada registro ha sido utilizado para operaciones de lectura, escritura, o lectura/escritura; y también, el *tiempo de vida* de cada registro, es decir, el número de ciclos de reloj que un registro almacena información útil para el programa. Cualquier fallo que ocurra en el registro durante este tiempo destruye la integridad del dato almacenado [Lee and Shrivastava, 2009a].

El *número de accesos* tiene un impacto considerable en el tamaño del código y la degradación del rendimiento, pues la protección de un registro que sea accedido muchas veces en el programa implica un mayor número de instrucciones adicionales. Por otro lado, el segundo parámetro, el *tiempo de vida*, está relacionado con la confiabilidad del programa, ya que cuanto más grande sea el *tiempo de vida* de un registro, mayor tiempo en el que éste sea susceptible de sufrir un fallo. Por consiguiente, ambos parámetros se deben tener en cuenta a la hora de seleccionar la estrategia de protección de registros en el software.

La Tabla 5.2 presenta el informe de utilización de recursos, proporcionado por el ISS, para el programa FIR. Nótese que en este programa sólo se utilizan los registros: 0, 1, 2, 3, y F. El *tiempo de vida* de los registros está expresado como el porcentaje del tiempo total del programa. Como se puede apreciar, el registro que presenta el *tiempo de vida* más alto es el F (100.0 % de los ciclos de reloj que dura el programa). Este mismo registro, a su vez, es el menos accedido; y consecuentemente, es el primer candidato para ser endurecido por software.

Cuadro 5.2: Utilización de registros en el programa FIR

Registro	# Escrituras	# Lecturas	# Lectura/Escritura	Tiempo de vida [%]
0	2806	3320	0	34.81 %
1	1785	765	8160	75.16 %
2	1276	1531	10841	88.27 %
3	1276	1530	8925	88.27 %
F	1	256	255	100.0 %

Como en el caso anterior, el programa se ha endurecido mediante *SWIFT-R selectivo* y, con fines ilustrativos, se han generado las 32 versiones posibles del software del filtro FIR (incluyendo la versión no-endurecida). La Figura 5.24 representa los incrementos en el tamaño del código fuente y tiempo de ejecución.

Los resultados están normalizados con respecto a una línea base construida con el tamaño del código y el tiempo de ejecución de la versión no-endurecida.

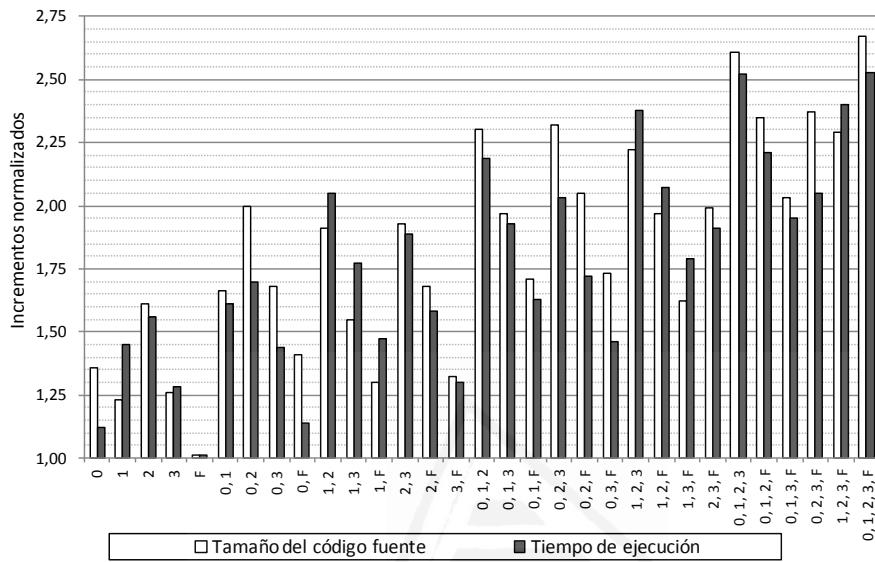


Figura 5.24: Impacto del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones del FIR (datos normalizados)

La confiabilidad de cada una de las diferentes configuraciones hardware/software del sistema se evaluó mediante campañas de inyección de fallos en *FT-Unshades* (con las mismas características que en el caso anterior). La Figura 5.25 representa los resultados obtenidos tras llevar acabo las campañas de test. Se presentan los porcentajes de clasificación de fallos obtenidos en cada versión del sistema (todas las versiones del software ejecutándose en cada una de las diferentes versiones del hardware). El AVF de cada sistema se calcula como la suma del porcentaje de fallos clasificados como SDC y Hang.

A partir del análisis conjunto de la Tabla 5.2 (utilización de registros), la Figura 5.24 (*overheads*) y la Figura 5.25 (cobertura frente a fallos), se puede concluir lo siguiente:

- Cuando se aplica *SWIFT-R selectivo* a los registros que son altamente accedidos, como el registro 2 (1276 veces para operaciones de escritura, 1531 veces para lecturas, y 10841 veces para operaciones de lectura/escritura), los *overheads* de tamaño de código y tiempo de ejecución aumentan drásticamente ($\times 1.72$ y $\times 1.63$ respectivamente).
- El AVF de los registros está relacionado directamente con su *tiempo de vida*. Sin embargo, el uso que se hace de cada registro dentro de la apli-

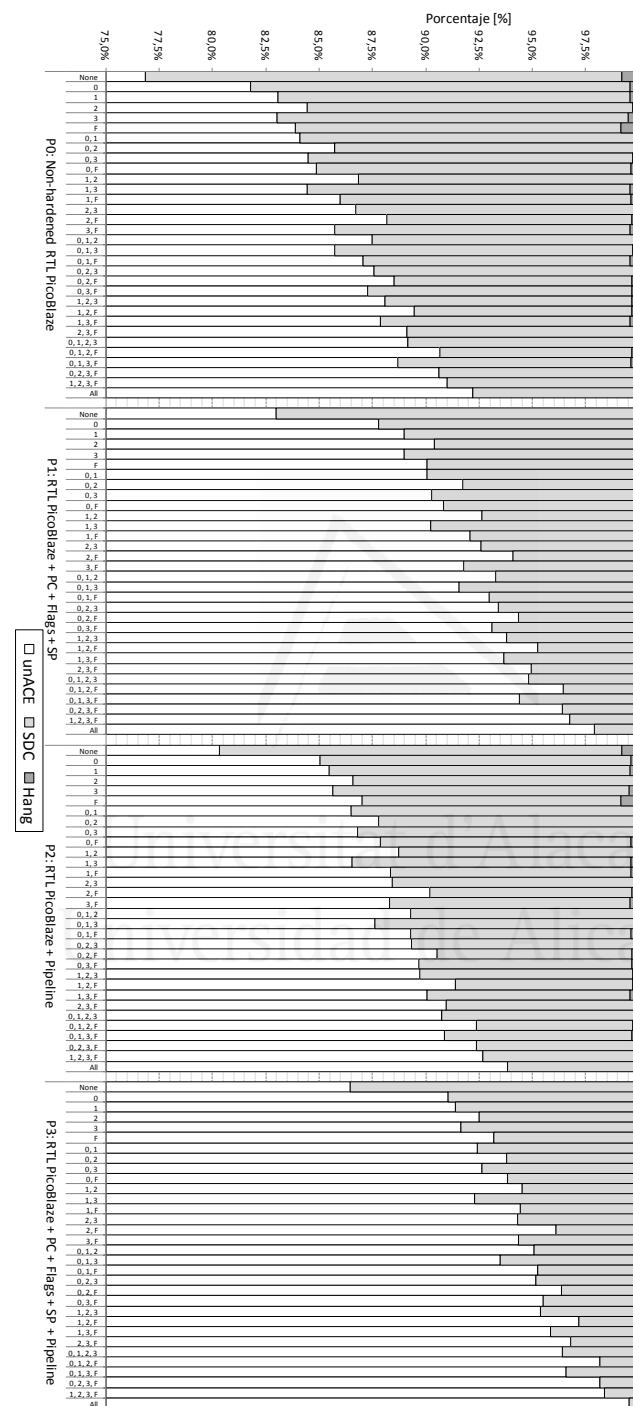


Figura 5.25: Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para el caso de la aplicación FIR

cación (las tareas que realiza en el programa) también influye en su nivel de vulnerabilidad.

- El AVF de los registros no siempre mantiene una correlación con el número de veces que el registro es accedido, por ejemplo, obsérvese el registro F , que es uno de los más vulnerables, es también el menos accedido (1 escritura, 256 lecturas, y 255 operaciones de lectura/escritura).

Se puede observar que la técnica SWIFT-R ofrece una mejora considerable de la cobertura frente a fallos, incluso para la versión del hardware sin endurecer, la cual alcanza hasta un 92.20 % de fallos unACE para el programa completamente endurecido ("All" en la Figura 5.25). Este porcentaje es más alto que cualquiera de las versiones del hardware endurecido utilizadas con el programa sin endurecer (las versiones "None" en la figura). Los resultados ofrecidos por la versión del hardware endurecido $P4$ no se presentan porque el 100.0 % de los fallos inyectados fueron clasificados como unACE, es decir, una cobertura completa frente a los fallos, tal cual como se esperaba.

De forma adicional, nótese que al combinar la protección software de la técnica SWIFT-R con el endurecimiento hardware de tan sólo unos pocos registros críticos, tales como: PC , $flags$, y SP (versión $P1$ del hardware); la cobertura frente a fallos se incrementa considerablemente (hasta alcanzar un 97.91 % de fallos clasificados como unACE).

Como se puede ver, al igual que en el caso anterior, la redundancia hardware en los registros internos del *pipeline* del microprocesador no se traduce en una mejora importante de la cobertura frente a fallos (versión $P2$ del microprocesador), incluso considerando que el número de registros protegidos en $P2$ es mayor que el número de estos en $P1$, cuya cobertura frente a fallos es más alta. Esto se debe a que los registros internos del *pipeline* no son tan críticos para el sistema, como sí lo son, por ejemplo, el PC y el SP .

A continuación, se deben analizar los *overheads* de tamaño de código y tiempo de ejecución de los programas de forma conjunta con los resultados de cobertura frente a fallos. Este proceso se debe llevar a cabo teniendo en cuenta los requisitos de la aplicación que se esté diseñando, lo cual es una de las principales claves durante el co-diseño del sistema. Estos análisis facilitan encontrar soluciones intermedias que posean mejores niveles de compromiso entre *overheads* y cobertura frente a fallos. Por ejemplo, la versión del software con SWIFT-R aplicado únicamente a los registros $\{0 - 3 - F\}$ ejecutándose en el microprocesador $P3$ sería una opción interesante, ya que por un lado ofrece alta cobertura frente a fallos (95.52 % de fallos unACE), y por otra parte, ocasiona incrementos aceptables del tamaño del código y del tiempo de ejecución del programa ($\times 1.73$ y $\times 1.46$ respectivamente).

Hay que tener en cuenta que aunque se pueden obtener grandes mejoras en la confiabilidad de los sistemas al combinar la protección software con los diferentes enfoques de endurecimiento hardware (por ejemplo, hasta 99.52 % de fallos unACE en la versión $P3$), el coste del hardware también se incrementa considerablemente. Este es un hecho importante a considerar cuando se explora el espacio de diseño. Como se ha mencionado antes, en este trabajo los costes

hardware se han estimado en términos de lógica secuencial (*flip-flops* y *latches*) y lógica combinacional (véase la Figura 5.18).

La Figura 5.26, por un lado, muestra los costes hardware de cada versión endurecida del microprocesador normalizados con respecto a una línea base construida con los datos de la versión *RTL PicoBlaze (P0)*; por otra parte, esta figura también ilustra, en un eje secundario, el MWTF de cada uno de los sistemas híbridos normalizado también con respecto a una línea base construida con la versión hardware/software no-endurecida. Como el MWTF captura el balance existente entre la cobertura frente a fallos y el rendimiento, esta figura permite observar de un vistazo la representación de diversos niveles de compromiso entre confiabilidad, tiempo de ejecución, y costes hardware para cada una de las combinaciones hardware/software.

Al igual que para la figura anterior, la versión completamente endurecida del microprocesador (*P4*) no está representada debido a su alto coste hardware en comparación con *P0*: $2.92\times$ lógica secuencial, y $1.93\times$ lógica combinacional. En este caso, los altos costes hardware pueden resultar poco adecuados para muchas aplicaciones de los sistemas embebidos, aunque su confiabilidad es 100%, esto significa que 100% de los fallos inyectados no causaron ningún efecto inesperado en el comportamiento del sistema (fallos unACE).

Resulta importante observar que los costes hardware crecen considerablemente cuando se protegen los registros del *pipeline* (microprocesadores *P2* y *P3*), mientras que en sus MWTF normalizados se puede observar apenas una ligera mejoría (casi igual a los resultados de la versión del hardware sin endurecer), o incluso se pueden observar algunos casos donde se obtienen peores resultados si se compara con enfoques menos costosos (*P2+SWIFT-R*).

De los resultados de la Figura 5.26 se puede observar también que los mayores incrementos en el MWTF ocurren cuando los microprocesadores ejecutan la versión completamente endurecida del software (versiones *All* en la figura), cuyos MWTF normalizados son: $1.17\times$, $4.36\times$, $1.48\times$, y $19.09\times$ para los microprocesadores *P0*, *P1*, *P2*, y *P3* respectivamente. Además, nótese que aunque los registros que tienen protección hardware en el microporcesador *P2* (*pipeline* protegido) son más numerosos que aquellos protegidos en la versión *P1*, esta cantidad no implica mayor confiabilidad, ya que protegiendo tan sólo algunos registros críticos se puede obtener una gran mejora en el MWTF (como en el caso de *P1*).

Existen algunas configuraciones hardware/software protegidas mediante enfoques selectivos que resultan interesantes para resaltar en esta discusión. Por ejemplo, aquellas versiones del software con protección SWIFT-R en los registros {0 – 2 – 3 – *F*}, cuyos MWTF normalizados son: $1.20\times$, $3.14\times$, $1.48\times$, y $6.13\times$ para los microprocesadores *P0*, *P1*, *P2*, y *P3* respectivamente. Estas soluciones intermedias pueden servir como candidatos adecuados para aplicaciones de los sistemas embebidos en las cuales no sólo la confiabilidad sea el parámetro más importante, sino que también existan otros factores que deban cumplir con estrictas restricciones, relacionados con el tiempo de ejecución, por ejemplo.

Algunas veces la protección de todos los registros del banco mediante un

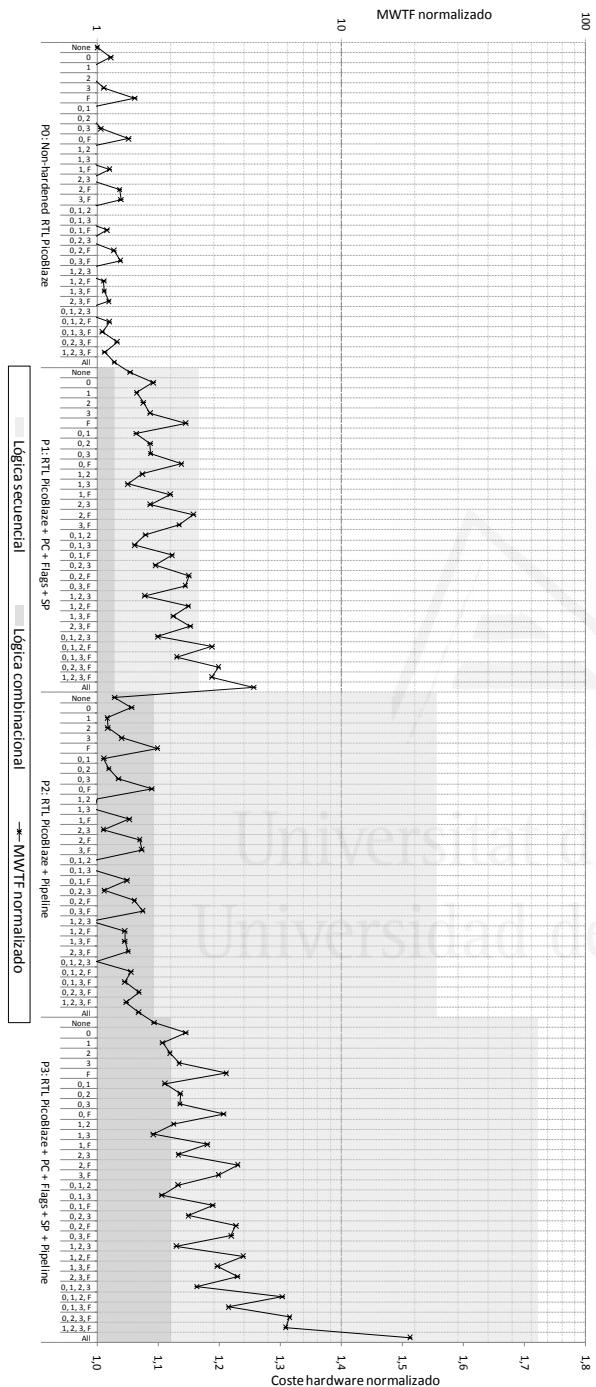


Figura 5.26: Costes hardware y MTTF para cada sistema endurecido selectivamente en hardware y software para el caso de la aplicación FIR

enfoque de endurecimiento software puede resultar en un sistema con una muy buena cobertura frente a fallos, pero al mismo tiempo esto provoca una degradación del rendimiento del programa inadecuada. Este hecho reduce el MWTF del sistema comparado con alguna versión protegida de forma selectiva, la cual causa una menor degradación del tiempo de ejecución. Esto se puede observar en la Figura 5.26 para los microprocesadores P_0 y P_2 cuando ejecutan las versiones del software: $\{F\}$ y $\{0 - F\}$.

Haciendo este tipo de análisis, se puede dirigir la exploración del espacio de diseño y encontrar una solución a la medida que satisfaga de la mejor manera las restricciones específicas y los requisitos de confiabilidad de cada aplicación. Para esta aplicación del filtro FIR, por ejemplo, podría resultar como una configuración adecuada del sistema aquella compuesta por el microprocesador P_1 ejecutando la versión del software protegida mediante SWIFT-R en los registros $\{0 - 2 - 3 - F\}$, ya que esta configuración del sistema ofrece un incremento importante en el tiempo medio hasta la avería ($3.14 \times$ más MWTF que el sistema sin protección), y a su vez, conlleva un bajo coste hardware y un incremento aceptable del tiempo de ejecución ($2.05 \times$). En otras aplicaciones, con menos restricciones en cuanto al coste hardware, la versión del software completamente endurecida con SWIFT-R ejecutándose en el microprocesador P_3 podría ser considerada como la mejor configuración del enfoque conjunto, ya que alcanza hasta $19.09 \times$ más MWTF que el sistema embebido sin endurecer.

Estos resultados se encuentran publicados en [Martínez-Álvarez et al., 2011].

Multiplicación de matrices (*mmult*)

En este tercer caso, se presenta un ejemplo del *co-endurecimiento* guiado por la aplicación. Se trata de un programa que implementa el algoritmo de multiplicación de matrices (*mmult*), y que se ejecuta en el microprocesador *RTL Pi-coBlaze*. La estrategia de protección es la misma que en los dos casos anteriores (*TMR selectivo* en hardware + *SWIFT-R selectivo* en software).

La Tabla 5.3 presenta el informe de utilización de recursos (*número de accesos* y *tiempo de vida* de cada registro) utilizados por el programa *mmult*. Nótese que en este programa sólo se utilizan los registros: 0, A, D, E, y F.

Cuadro 5.3: Utilización de los registros usados en el programa *mmult*

Registro	# Escrituras	# Lecturas	# Lectura/Escritura	Tiempo de vida [%]
0	340	357	183	55.3 %
A	20	83	45	85.7 %
D	27	135	108	48.7 %
E	27	135	108	52.2 %
F	9	9	27	83.3 %

De la Tabla 5.3 se puede ver que los registros que presentan un mayor *tiempo de vida* son A y F, con 85.7% y 83.3% del total de ciclos de reloj del programa respectivamente. Además, esos mismos registros son los menos accedidos, y

por lo tanto, su protección no influye de forma significativa en los incrementos del tamaño del código y el tiempo de ejecución del programa (como se puede ver en la Figura 5.27). En consecuencia, estos dos registros son los primeros candidatos a ser endurecidos en software. El siguiente registro con mayor *tiempo de vida* es el 0 (55.3 %), sin embargo, también presenta el mayor número de accesos por lo que su protección tendrá un impacto significativo sobre los *overheads* (como lo confirman los datos de la Figura 5.27). Por último, los registros *D* y *E* presentan un número de accesos menor que los del registro 0 y, por tanto, su influencia sobre los *overheads* será menor. Asimismo, como estos registros presentan el menor *tiempo de vida*, también se puede adelantar una menor incidencia en la cobertura frente a fallos al protegerlos.

La Figura 5.27 muestra los incrementos, en términos de tamaño de código fuente y tiempo de ejecución, causados al aplicar la técnica SWIFT-R a varios subconjuntos de registros del microprocesador. A diferencia de los dos casos anteriores (PID y FIR), en esta ocasión no se exploran todas las posibles combinaciones de registros endurecidos, sino que se parte de una pre-selección (en total 17 versiones del software diferentes, incluyendo la versión no endurecida). Esta pre-selección de versiones software se hace con el fin de ilustrar que el diseñador puede agilizar el proceso, al descartar algunas versiones del software endurecido que no resulten de interés. Los resultados están normalizados con respecto a una línea base construida con la versión no endurecida.

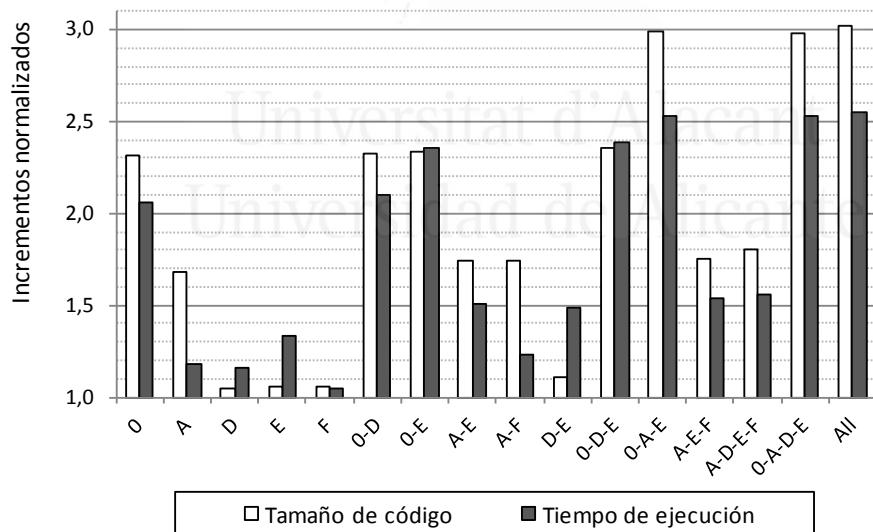


Figura 5.27: Impacto normalizado del endurecimiento selectivo sobre el tamaño del código fuente y el tiempo de ejecución de las distintas versiones endurecidas del programa *mmult*

En este caso, se han utilizado las mismas 5 versiones hardware del micro-

procesador para complementar la estrategia de protección, éstas son: *P0*, *P1*, *P2*, *P3*, y *P4*. Se han implementado en total 69 versiones distintas del sistema: 17 versiones del software corriendo en los microprocesadores *P0*, *P1*, *P2* y *P3*, más el programa sin endurecer ejecutándose en el hardware completamente endurecido (*P4*).

Para evaluar la confiabilidad de cada una de las configuraciones hardware/software del sistema se realizó una campaña de inyección de fallos en *FT-Unshades* (con las mismas características que antes). La Figura 5.28 presenta los porcentajes de clasificación de fallos obtenidos en el experimento.

Nótese que una vez más, la técnica SWIFT-R ofrece, por sí sola, una mejora considerable en la cobertura frente a fallos, incluso en las versiones del hardware sin endurecer, alcanzando un 95.38 % de fallos unACE en el programa completamente endurecido, el cual es un porcentaje incluso mayor que los resultados obtenidos por cualquier versión del hardware endurecido ejecutando el software sin protección. Los resultados del microprocesador *P4* no se presentan en la Figura 5.28 porque el 100 % de los fallos inyectados fueron clasificados como unACE, como se esperaba. Además, una vez más, se puede observar que al combinar SWIFT-R con protección hardware sólo los registros críticos, como: *PC*, *Flags*, y *SP* (microprocesador *P1*), se logra obtener una mejora importante en la cobertura frente a fallos (hasta 98.32 % fallos unACE).

En este caso de estudio, se obtiene uno de los resultados interesantes cuando SWIFT-R se aplica de forma selectiva al subconjunto de registros {*A*, *D*, *E*, *F*} y se ejecuta en el microprocesador *P3*. Esta configuración del sistema ofrece, por un lado, alta cobertura frente a fallos (98.68 % unACE), y por otra parte, incrementos muy contenidos en tamaño del código y tiempo de ejecución ($\times 1.80$ y $\times 1.56$ respectivamente).

En cuanto al análisis combinado de los resultados, la Figura 5.29 muestra el porcentaje de fallos clasificados como unACE en cada una de las diferentes configuraciones hardware/software del sistema; y además, esta figura presenta también, en un eje secundario, los costes del endurecimiento (costes hardware e incremento en el tiempo de ejecución) para cada versión del sistema. Estos costes se encuentran normalizados con respecto a la versión no endurecida del sistema (*RTL PicoBlaze P0* ejecutando la versión del programa sin endurecer).

Se observa que los costes hardware se elevan considerablemente cuando se protegen los registros del *pipeline* (*P2* y *P3*), mientras que en estos casos la cobertura frente a fallos apenas mejora levemente, o incluso disminuye si se compara con enfoques menos costosos (*P2+SWIFT-R*). En el caso de *P4*, los elevados costes hardware pueden determinar que éste sea un enfoque poco viable en muchas aplicaciones, aún cuando su cobertura frente a fallos de este tipo es del 100 %. Con el fin de establecer comparaciones, se debe notar que esta solución tiene aproximadamente un incremento de área hardware de $\times 3$ más y una degradación despreciable en el rendimiento (en comparación con las alternativas del software endurecido).

La exploración de las diferentes soluciones de compromiso resultantes entre rendimiento, confiabilidad, tamaño de código, y coste hardware, permiten al diseñador decidir cual de todas las posibles configuraciones del sistema sa-

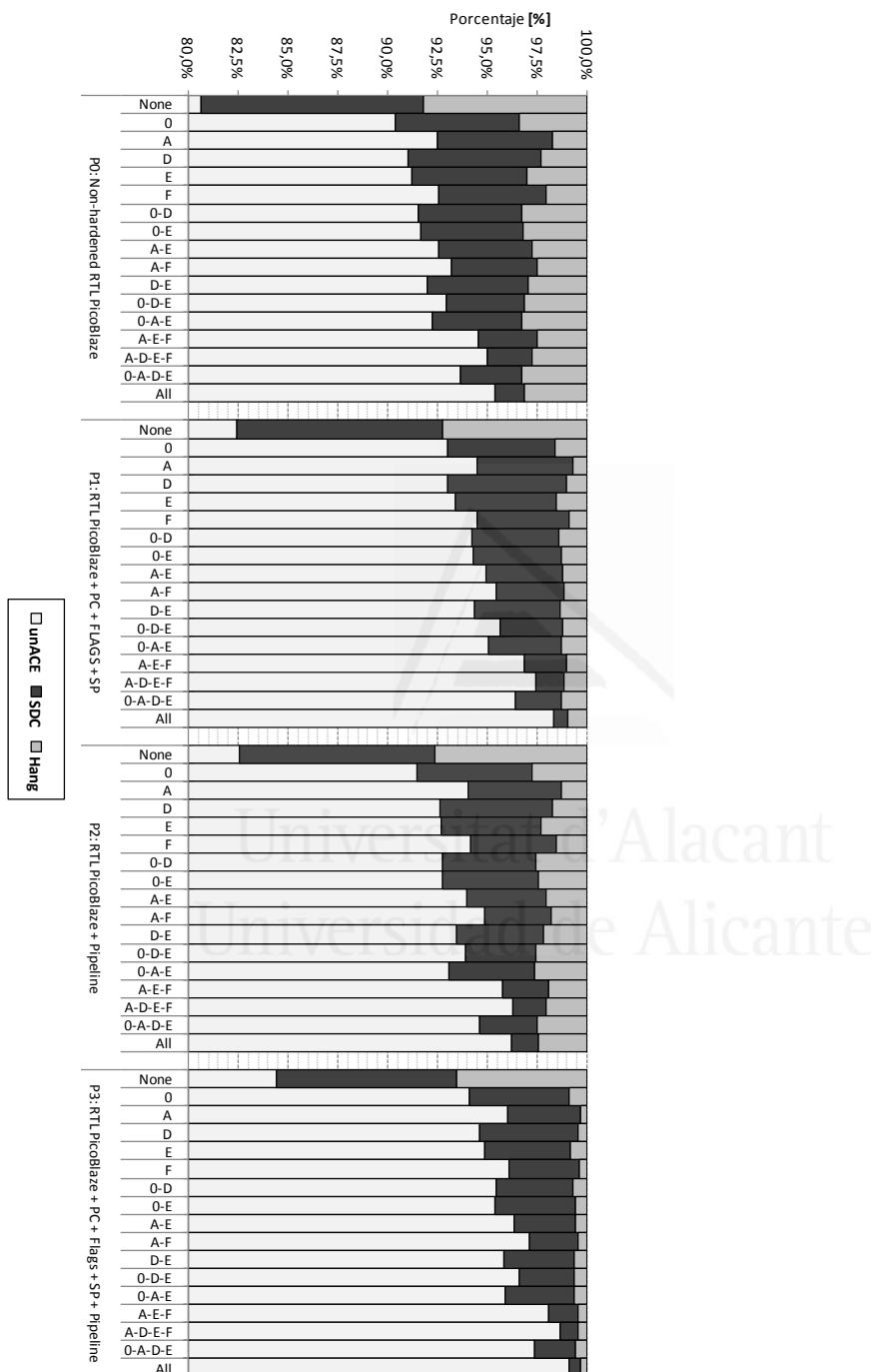


Figura 5.28: Porcentajes de clasificación de fallos para cada sistema endurecido selectivamente en hardware y software para la aplicación *mmult*

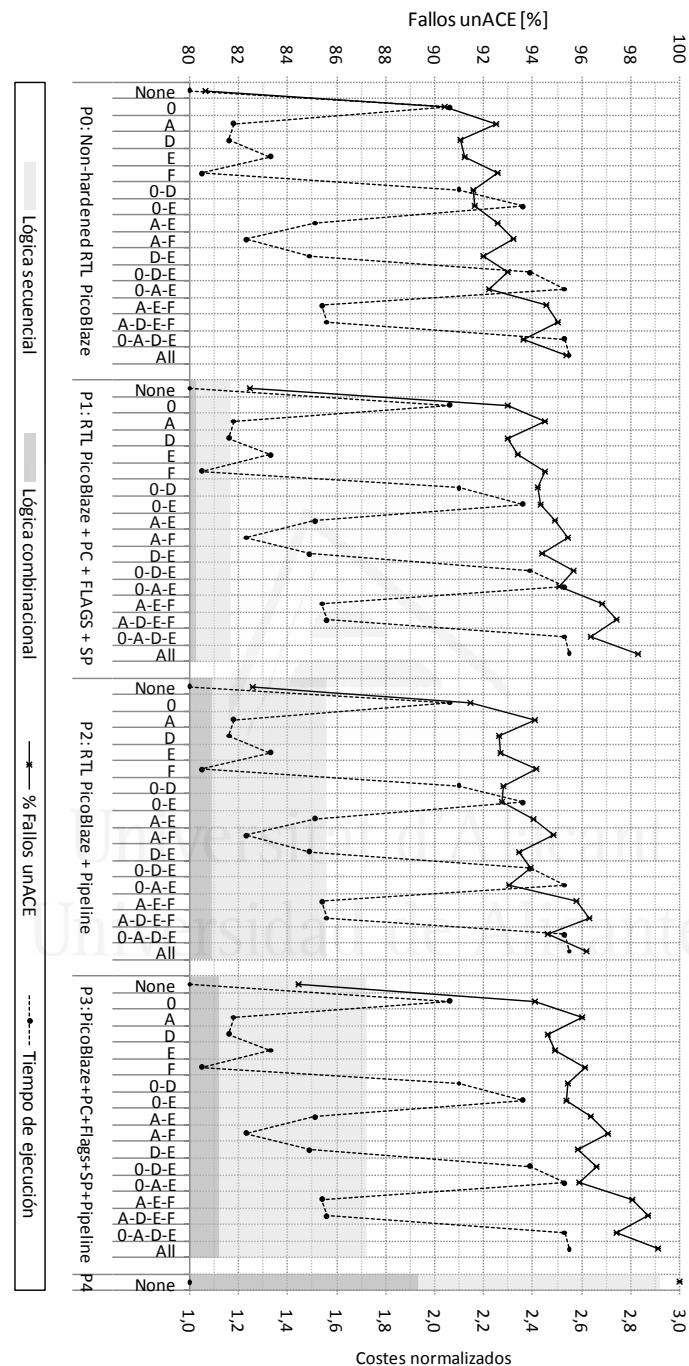


Figura 5.29: Porcentajes de fallos unACE y costes de endurecimiento normalizados (coste hardware e incremento del tiempo de ejecución) para cada versión del sistema para la aplicación *mmult*

tisface mejor sus requisitos. Por ejemplo, en este caso, merece la pena resaltar la versión formada por el hardware *P1* y SWIFT-R aplicado a los registros $\{A - D - E - F\}$ en el software, porque ofrece un nivel alto de cobertura frente a fallos (97.43 % fallos unACE) con costes y *overheads* aceptables.

Estos resultados se encuentran publicados en [Cuenca-Asensi et al., 2011a].

5.4. Conclusiones

En este capítulo se ha presentado un completo caso de estudio donde se ha podido verificar la flexibilidad y la aplicabilidad del *co-endurecimiento* para el co-diseño de sistemas hardware/software tolerantes a fallos inducidos por radiación.

En primer lugar, se ha seleccionado el microprocesador *PicoBlaze* para llevar a cabo el caso de estudio. Para esto, se ha desarrollado un *front-end* y un *back-end* para integrar dicho microprocesador dentro de SHE; además, en cuanto al hardware del sistema, se ha diseñado una versión del microprocesador independiente del fabricante, que posee las mismas características y es RTL equivalente a la versión original de *Xilinx* (*RTL PicoBlaze*). La selección de este microprocesador se fundamenta en que además de que es simple y altamente portable o trasladable a distintas tecnologías de fabricación de circuitos, al ser un *soft core* ofrece la plasticidad necesaria para poder modificar los componentes internos del hardware en caso que sea necesario al aplicar alguna técnica de endurecimiento hardware. Además, también tiene limitaciones importantes en cuanto a rendimiento y recursos físicos, lo cual hace que sea necesario el co-diseño hardware/software al implementar estrategias de tolerancia a fallos para sus aplicaciones.

Con el fin de verificar la flexibilidad y la utilidad del entorno de desarrollo de endurecimiento software presentado (SHE), se ha realizado el diseño de tres técnicas de tolerancia a fallos basadas en software: *TMR1*, *TMR2*, y *SWIFT-R*. Asimismo, se han evaluado dichas técnicas mediante su aplicación a un conjunto representativo de programas de prueba donde se ha tenido en cuenta lo siguiente: se ha verificado que los programas endurecidos son equivalentes funcionalmente a sus versiones originales; se ha estudiado el impacto que causa el endurecimiento en el tamaño del código y el rendimiento de los programas; y también, se ha evaluado la confiabilidad proporcionada por cada uno de los tres enfoques. La confiabilidad de los sistemas se ha estimado a partir de la realización de campañas de inyección de fallos basadas en simulación de fallos tipo SEU (en el ISS) y emulación de fallos (en *FT-Unshades*).

Se ha verificado la aplicabilidad del enfoque propuesto, para lo cual se ha diseñado una estrategia de *co-endurecimiento* hardware/software que permite encontrar con facilidad el punto óptimo en el espacio de diseño de los sistemas embebidos tolerantes a fallos.

La técnica de *co-endurecimiento* desarrollada combina el endurecimiento selectivo del software y del hardware: por parte del software se ha implementado y mejorado la técnica conocida como SWIFT-R, mientras que la protección

hardware busca complementar la anterior y se basa en la aplicación de triple redundancia modular a distintos conjuntos de registros internos del microprocesador.

Esta técnica ha sido aplicada a tres aplicaciones distintas de los sistemas embebidos con el fin de validar el enfoque de forma experimental. Estas aplicaciones fueron: filtro FIR, controlador PID, y multiplicación de matrices. Los resultados evidencian que mediante el *co-endurecimiento* es posible realizar con facilidad la exploración del espacio de diseño y encontrar una solución a la medida que satisfaga de la mejor manera las restricciones específicas y los requisitos de confiabilidad propias de cada aplicación. Se ha evidenciado también, que no siempre la mejor configuración hardware/software del sistema es la que conlleva un mayor nivel de redundancia, y muchas veces las versiones protegidas mediante el endurecimiento selectivo únicamente de sus componentes críticos pueden resultar más adecuadas, ya que es un amplio conjunto de parámetros el que se debe tener en cuenta al evaluar las bondades del sistema, entre ellos: confiabilidad, rendimiento, tamaño código fuente y costes hardware.

Además, se ha podido comprobar que la información que ofrecen nuestras herramientas guía al diseñador en la exploración del espacio de soluciones y facilita la reducción del área a explorar.

Capítulo 6

Conclusiones y trabajos futuros

Este capítulo final reúne las conclusiones finales del trabajo realizado en esta tesis doctoral. El capítulo está organizado en cuatro secciones. La Sección 6.1 presenta las principales conclusiones generales del trabajo de investigación llevado a cabo. Seguidamente, en la Sección 6.2 se resumen las principales aportaciones científicas de esta tesis. A continuación, en la Sección 6.3 se presenta el listado de las publicaciones científicas consecuencia del trabajo realizado. Por último, la Sección 6.4 plantea los problemas abiertos e identifica las líneas de trabajo futuro para continuar con la investigación.

6.1. Conclusiones generales

Durante las últimas décadas, la miniaturización de los componentes electrónicos ha ocasionado que los sistemas digitales sean cada vez más susceptibles a fallos transitorios inducidos por radiación. Este hecho ha causado que la importancia de diseñar sistemas digitales tolerantes a este tipo de fallos haya ido en aumento.

A pesar de que los fallos inducidos por radiación eran considerados usualmente como un problema propio de los sistemas de misión crítica que operan a nivel espacial, desde hace ya varios años, este problema se ha visto trasladado también a los sistemas confiables desplegados a nivel atmosférico e incluso a nivel terrestre. Esto hace aún más evidente la necesidad de mitigar los efectos de la radiación en los circuitos electrónicos.

Las técnicas de diseño para la mitigación de fallos están basadas en la incorporación de módulos redundantes en el sistema. Por un lado, los enfoques de diseño basados en redundancia hardware ofrecen una solución efectiva para la mitigación de fallos. Sin embargo, como suponen principalmente la adición de nueva lógica hardware para satisfacer los requisitos de confiabilidad, se emplean más componentes electrónicos, se consume más potencia, se utiliza más

área, el coste económico es mayor, y se incrementa el tiempo de desarrollo. Por consiguiente, estos enfoques no resultan adecuados en muchos casos. Por otro lado, gracias al auge de los sistemas digitales basados en microprocesador y con el ánimo de buscar soluciones menos costosas, han surgido numerosas propuestas de mitigación de fallos basadas en redundancia software. Aunque estos enfoques están libres de costes hardware y permiten la utilización de componentes COTS, su utilización perjudica el rendimiento de los programas y aumenta el tamaño del código fuente.

Se han podido identificar evidencias de que mediante la utilización de técnicas de protección híbridas hardware/software es posible encontrar un punto medio entre las soluciones hardware y software donde se aprovechan las bondades que ofrecen ambos enfoques por separado. Sin embargo, las técnicas híbridas existentes actualmente son muy específicas y carecen de la flexibilidad necesaria para encontrar los mejores niveles de compromiso entre fiabilidad, rendimiento, coste, y otros parámetros asociados a las restricciones del diseño y los requisitos de confiabilidad de cada aplicación.

En este trabajo se ha propuesto y llevado a cabo el diseño de sistemas bebidos tolerantes a fallos inducidos por radiación mediante lo que se ha denominado: *co-endurecimiento*. Se basa en los principios del co-diseño para diseñar a medida una estrategia híbrida de mitigación de fallos, basada en la combinación selectiva, incremental y flexible de enfoques hardware y software. La exploración del espacio de diseño se fundamenta en una estrategia híbrida de grano fino. De esta forma, es posible diseñar sistemas confiables a bajo coste, donde no sólo se satisfacen los requisitos de confiabilidad y las restricciones de diseño, sino que también se evita el uso excesivo de costosos mecanismos de protección (hardware y software).

Se ha propuesto un flujo de diseño para el *co-endurecimiento* que está guiado por las especificaciones de la aplicación que se esté diseñando y, además, se ha propuesto priorizar los enfoques de endurecimiento software teniendo en cuenta los elevados tiempos de diseño, implementación y evaluación de las técnicas hardware. De esta forma, la estrategia de endurecimiento software, es complementada en una etapa posterior, mediante mecanismos hardware de redundancia. Este método de trabajo agiliza el tiempo de diseño necesario para la estrategia de protección y, combinado con la aplicación selectiva de técnicas software y hardware, permite encontrar soluciones quasi-óptimas (optimización del espacio de co-diseño).

El *co-endurecimiento* es aplicable a tecnologías ASIC o FPGA para la implementación final. En el caso de las FPGAs, se evita tener que utilizar las costosas FPGAs *rad-hard* ya que podrían utilizarse dispositivos *Flash* que implementen la tolerancia a fallos mediante técnicas de diseño. No obstante, para utilizar FPGAs SRAM se requieren mecanismos de protección adicional para la memoria de configuración.

Por otra parte, para soportar el *co-endurecimiento*, se ha presentado una infraestructura que permite el diseño, implementación y evaluación de las estrategias híbridas de tolerancia a fallos. Esta infraestructura está formada por dos herramientas: un entorno de endurecimiento software (*Software Hardening En*

vironment - SHE), y una herramienta para evaluar la confiabilidad ofrecida por las técnicas híbridas diseñadas (*FT-Unshades*).

El entorno desarrollado para el endurecimiento software (SHE) es el encargado de apoyar al diseñador y guiar el proceso de *co-endurecimiento*, al establecer una primera aproximación a la partición hardware/software del sistema global. Esta herramienta permite llevar a cabo las siguientes tareas:

- Diseño de técnicas software de tolerancia a fallos basadas en la redundancia de instrucciones a nivel de código ensamblador.
- Aplicación de las técnicas diseñadas de forma automática en el código fuente de los programas.
- Endurecimiento de software para múltiples microprocesadores diferentes de forma independiente de la arquitectura. Esto es posible gracias a que el entorno se fundamenta en un *núcleo de endurecimiento genérico* basado en una *arquitectura genérica de microprocesadores*.

Además, SHE se caracteriza por:

- Ofrecer una interfaz de programación del endurecimiento software (API) extensible que provee los métodos más comunes empleados en la programación de técnicas software de tolerancia a fallos.
- Facilitar el diseño de técnicas software que pueden ser aplicadas de forma completa o de forma selectiva a los diferentes recursos/componentes del programa. Es decir, hace posible el diseño de técnicas software que se apliquen sólo a las partes/recursos más críticos desde el punto de vista software.
- Permitir evaluar de forma preliminar la confiabilidad proporcionada por las técnicas diseñadas.

Por otra parte, *FT-Unshades* permite realizar la evaluación de la confiabilidad. Está compuesta por una placa hardware de emulación de fallos basada en FPGA y un conjunto de herramientas software que permiten comprobar el diseño, recoger y analizar los resultados de las campañas de inyección de fallos. Sus principales ventajas son:

- Permite la inyección de fallos de forma no intrusiva, es decir, sin modificar el diseño original.
- Otorga flexibilidad en el tiempo para que el circuito en el cual se emulen los fallos tenga un tiempo determinado para recuperarse y volver a un estado libre de errores. Esto permite realizar análisis sobre sistemas basados en microprocesador en los que se hayan implementado técnicas software de endurecimiento.
- Ofrece la posibilidad de realizar análisis jerárquicos, donde es posible estudiar por separado la confiabilidad de cada uno de los módulos internos que conforman el sistema.

- Permite el acceso a todos los recursos físicos del sistema (al tratarse de una implementación real), incluso a nivel de la microarquitectura. Esto facilita la obtención de resultados de confiabilidad más realistas que los que se pueden obtener por medio de técnicas de inyección de fallos basadas en simulación.
- Permite evaluar las técnicas de endurecimiento basadas en software y hardware.

Mediante el enfoque propuesto y las herramientas que lo soportan, se ha llevado a cabo un completo caso de estudio donde se ha podido verificar la flexibilidad y la aplicabilidad del *co-endurecimiento* para el co-diseño de sistemas hardware/software tolerantes a fallos inducidos por radiación.

En primer lugar, para verificar la flexibilidad y la utilidad del entorno de desarrollo de endurecimiento software presentado (SHE), se han diseñado tres técnicas de tolerancia a fallos basadas en software: *TMR1*, *TMR2*, y *SWIFT-R*. Dichas técnicas han sido evaluadas mediante su aplicación a un conjunto de programas de prueba donde se ha tenido en cuenta lo siguiente: verificación funcional de los programas endurecidos con respecto a los programas originales; análisis del impacto del endurecimiento en el tamaño del código y el rendimiento de los programas; y por último, evaluación de la confiabilidad proporcionada por cada uno de los tres enfoques.

En segundo lugar, se ha verificado la aplicabilidad del enfoque propuesto, para lo cual se ha diseñado una estrategia de *co-endurecimiento* hardware/software que permite encontrar con facilidad un punto óptimo en el espacio de diseño de los sistemas embebidos tolerantes a fallos. La técnica de *co-endurecimiento* desarrollada combina el endurecimiento selectivo del software y del hardware: por parte del software se ha implementado y mejorado la técnica conocida como *SWIFT-R*, mientras que la protección hardware busca complementar la anterior y se basa en la aplicación de TMR a distintos conjuntos de registros internos del microprocesador.

La metodología para el *co-endurecimiento* ha sido aplicada de forma exhaustiva a tres aplicaciones distintas de los sistemas embebidos con el fin de validar el enfoque de forma experimental. Estas aplicaciones fueron: controlador PID, filtro FIR y multiplicación de matrices. De los resultados obtenidos se puede concluir que mediante el *co-endurecimiento* se facilita la exploración del espacio de diseño y es posible encontrar una solución a la medida que satisface, de la mejor manera posible, las restricciones específicas y los requisitos de confiabilidad propias de cada aplicación. También, es importante tener en cuenta que no siempre la mejor configuración hardware/software del sistema es la que lleva un mayor nivel de redundancia, y muchas veces las versiones protegidas mediante el endurecimiento selectivo únicamente de sus componentes críticos pueden resultar más adecuadas.

6.2. Aportaciones

En este contexto, merece la pena mencionar de forma puntual las aportaciones más importantes que se desprenden del trabajo realizado en esta tesis doctoral:

1. Propuesta de un metodología de *co-endurecimiento*, basada en los principios del co-diseño de sistemas, para el desarrollo de estrategias híbridas hardware/software de tolerancia a fallos.
2. Desarrollo de una infraestructura flexible para el endurecimiento de sistemas que soporta el enfoque de *co-endurecimiento* y asiste al diseñador en el diseño, implementación y evaluación de técnicas de tolerancia a fallos, y por consiguiente, facilita la exploración del espacio de diseño.
3. Propuesta e implementación de una nueva versión selectiva de la técnica de recuperación de fallos conocida como SWIFT-R.

6.3. Publicaciones

A lo largo del desarrollo de la presente tesis doctoral, y en relación con los resultados y aportaciones de la misma, se han realizado las siguientes publicaciones científicas:

- Artículos en revistas científicas de alto impacto:
 - [Martínez-Álvarez et al., 2011] Martínez-Álvarez, A., Cuenca-Asensi, S., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H., and Aguirre, M. A. (2011). Compiler-directed soft error mitigation for embedded systems. *IEEE Transactions on Dependable and Secure Computing*, In Press.
 - [Cuenca-Asensi et al., 2011b] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H., and Aguirre, M. A. (2011). Soft core based embedded systems in critical aerospace applications. *Journal of Systems Architecture*, In Press, Corrected Proof.
 - [Cuenca-Asensi et al., 2011a] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F., Guzmán-Miranda, H., and Aguirre, M. (2011). A novel co-design approach for soft errors mitigation in embedded systems. *IEEE Transactions on Nuclear Science*, 58(3):1059-1065.
- Conferencias científicas internacionales:
 - [Restrepo-Calle et al., 2011] Restrepo-Calle, F., Cuenca-Asensi, S., Aguirre, M. A., Palomo, F. R., Guzmán-Miranda, H., and Martínez-Álvarez, A. (2011). On the definition of real conditions for a fault injection

- experiment on embedded systems. In Proceedings of the 12th European Conference on Radiation and its Effects on Components and Systems RADECS 2011. Sevilla, Spain. Sept 19 - 23, 2011.
- [Lindoso et al., 2011] Lindoso, A., Entrena, L., San Millán, E., Cuenca-Asensi, S., Martínez-Álvarez, A., and Restrepo-Calle, F. (2011). A co-design approach for set mitigation in embedded systems. In Proceedings of the 12th European Conference on Radiation and its Effects on Components and Systems RADECS 2011. Sevilla, Spain. Sept 19 - 23, 2011.
 - [Restrepo-Calle et al., 2010g] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F. R., Guzmán-Miranda, H., Aguirre, M. A., and Cuenca-Asensi, S. (2010). A novel co-design approach for soft errors mitigation in embedded systems. In Proceedings of the 11th European Conference on Radiation and its Effects on Components and Systems RADECS 2010. Langenfeld, Austria. Sept 20 - 24, 2010.
 - [Restrepo-Calle et al., 2010f] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F., Guzmán-Miranda, H., Aguirre, M., and Cuenca-Asensi, S. (2010). Rapid Prototyping of Radiation-Tolerant Embedded Systems on FPGA. In Proceedings of the 20th International Conference on Field Programmable Logic and Applications FPL 2010, pages 326-331. Milano, Italy. Aug 31-Sep 2. 2010.
 - [Restrepo-Calle et al., 2010d] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., and Cuenca-Asensi, S. (2010). Application-driven co-design of fault-tolerant industrial systems. In Proceedings of the IEEE International Symposium on Industrial Electronics, ISIE. Bari, Italy. July 4-7. 2010.
 - [Restrepo-Calle et al., 2010b] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., Aguirre, M., and Cuenca-Asensi, S. (2010). A compiler-based infrastructure for fault-tolerant co-design. In Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems, SCOPES. Art. no. 4, St. Goar, Germany, 28-29 June 2010.
 - [Restrepo-Calle et al., 2010a] Restrepo-Calle, F., Martínez-Álvarez, A., Cuenca-Asensi, S., Palomo, F., and Aguirre, M. (2010). Hardening development environment for embedded systems. In the 2nd HiPEAC Workshop on Design for Reliability (DFR'10) held in conjunction with The 5th Int. Conf. on High Performance and Embedded Architectures and Compilers. Pisa, Italy, Jan 25-27, 2010.
- Conferencias científicas nacionales:
- [Restrepo-Calle et al., 2010e] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F., Guzmán-Miranda, H., Aguirre, M., and Cuenca-Asensi, S. (2010). Prototipado Rápido de Sistemas Empotrados Tolerantes a Radiación en FPGA. En Actas de las X Jornadas de Computación

Reconfigurable y Aplicaciones JCRA 2010, pages 261-268. Valencia, España, Septiembre 2010.

- [Restrepo-Calle et al., 2010c] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., Aguirre, M., and Cuenca-Asensi, S. (2010). Mitigación automática de soft-errors en nuevas aplicaciones de los sistemas empotrados. En Actas del I Simposio en Computación Empotrada, SiCE 2010, pages 25-32. Valencia, España, Septiembre 2010.

6.4. Trabajo futuro

En esta memoria se ha presentado un enfoque para el co-diseño de sistemas hardware/software tolerantes a fallos inducidos por radiación, el cual se ha denominado: *co-endurecimiento*. No obstante, se puede seguir profundizando e investigando alrededor de algunos aspectos que han quedado aún por desarrollar. A continuación se mencionan los más relevantes:

- Gracias a las ventajas y flexibilidad del entorno de endurecimiento software (SHE), será posible la exploración automática del espacio de diseño del lado software mediante la implementación de un algoritmo de optimización multi-objetivo, como por ejemplo NSGA-II [Deb et al., 2002]. Actualmente se han iniciado labores al respecto.
- A raíz de los resultados que se obtengan en el punto anterior, se investigará también la posibilidad de realizar la exploración del espacio de diseño hardware/software de forma automática.
- Extensión de SHE para soportar microprocesadores más complejos y potentes, que se utilicen en aplicaciones de misión crítica, como el procesador de 32-bits LEON3 [Gaisler, 2011] y el microcontrolador de 16-bits MSP430 [TexasInstruments, 2011]. De hecho, a día de hoy, se cuenta con más del 50 % del *front-end* de SHE desarrollado para éste último.
- Profundizar en el estudio de la relación existente entre, las estadísticas de acceso a registros y tiempo de vida, con el impacto en los *overheads* y el nivel de vulnerabilidad. La finalidad es poder obtener una métrica fiable que permita priorizar de forma automática y efectiva la protección de los distintos recursos del sistema.
- Diseño de nuevas técnicas de *co-endurecimiento* hardware/software utilizando la infraestructura presentada.
- Estudio de la efectividad del *co-endurecimiento* en la mitigación de otros tipos de fallos inducidos por radiación (diferentes a los fallos tipo SEU); principalmente los fallos tipo SET y MBU/MCU. Durante este último año, se han realizado avances significativos con respecto al estudio de fallos SET, gracias a la colaboración con el grupo de *Diseño Microelectrónico*

y Aplicaciones de la Universidad Carlos III de Madrid. Los resultados obtenidos han sido publicados recientemente en [Lindoso et al., 2011].

- Dentro de los objetivos del proyecto RENASER+¹ está prevista la realización de ensayos de irradiación en el Centro Nacional de Aceleradores (CNA) de algunos de los prototipos desarrollados. Estos ensayos permitirán evaluar, en un entorno real de irradiación, la efectividad de las técnicas de *co-endurecimiento*. Se prevé la realización de experimentos de TID (*Total Ionization Dose*), SEE (*Single Event Effects*) y NIEL (*Non Ionizing Energy Loss*) sobre los circuitos integrados.



Universitat d'Alacant
Universidad de Alicante

¹RENASER+: Análisis Integral de Circuitos y Sistemas Digitales para Aplicaciones Aeroespaciales (TEC2010-22095-C03-01)

Bibliografía

- [Abramovici et al., 1990] Abramovici, M., Breuer, M., and Friedman, A. (1990). *Digital System Testing and Testable Design*. IEEE Press, New York. 25
- [Actel, 2010] Actel (2010). RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs. Data Sheet Rev. 13, Actel Corporation. 50
- [Actel, 2011] Actel (2011). FPGA Solutions: Low Power FPGAs and Mixed Signal FPGAs. <http://www.actel.com/>. Internet. 29
- [AEC, 2003] AEC (2003). AEC-Q100-Rev-F Stress Test Qualification for Integrated Circuits. Technical report, Automotive Electronics Council. 4
- [Aguirre et al., 2005] Aguirre, M., Tombs, J., Munoz, F., Baena, V., Torralba, A., Fernandez-Leon, A., and Tortosa-Lopez, A. (2005). FT-UNSHADES: A new system for SEU injection, analysis and diagnostics over post synthesis netlist. In *In 8th Military and Aerospace Programmable Logic Device (MAPLD) International Conference*. 77, 99
- [Aguirre et al., 2004] Aguirre, M., Tombs, J. N., Muñoz, F., Baena, V., Torralba, A., Fernandez-Leon, A., Tortosa, F., and Gonzalez-Gutierrez, D. (2004). A FPGA based hardware emulator for the insertion and analysis of single event upsets in VLSI designs. In *Proc. Radiation Effects on Components and Systems Conf.*, page 1000. 55
- [Aguirre et al., 2007] Aguirre, M. A., Tombs, J. N., Munoz, F., Baena, V., Guzman, H., Napoles, J., Fernandez-Leon, A., Tortosa-Lopez, F., and Merodio, D. (2007). Selective protection analysis using a SEU emulator: testing protocol and case study over the LEON2 processor. *IEEE Transactions on Nuclear Science*, 54(4, Part 2):951–956. 102
- [Alderighi et al., 2010] Alderighi, M., Casini, F., D'Angelo, S., Mancini, M., Codinachs, D., Pastore, S., Poivey, C., Sechi, G., Sorrenti, G., and Weigand, R. (2010). Experimental Validation of Fault Injection Analyses by the FLIPPER Tool. *IEEE Transactions on Nuclear Science*, 57(4):2129 –2134. 54
- [Alderighi et al., 2003] Alderighi, M., Casini, F., D'Angelo, S., Mancini, M., Marmo, A., Pastore, S., and Sechi, G. (2003). A tool for injecting seu-like

- faults into the configuration control mechanism of xilinx virtex fpgas. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 71 – 78. 54
- [Altera, 2011] Altera (2011). FPGA CPLD and ASIC solutions from Altera. <http://www.altera.com/>. Internet. 29
- [Antoni et al., 2003] Antoni, L., Leveugle, R., and Feher, B. (2003). Using run-time reconfiguration for fault injection applications. *Instrumentation and Measurement, IEEE Transactions on*, 52(5):1468 – 1473. 55
- [Arif, 2010] Arif, T. (2010). *Aerospace Technologies Advancements*. Intech, Olajnica 19/2, 32000 Vukovar, Croatia. ISBN: 978-953-7619-96-1. 4
- [Austin, 1999] Austin, T. (1999). DIVA: A reliable substrate for deep submicron microarchitecture design. In *32nd Annual Int. Symp. on Microarchitecture, (MICRO-32)*, pages 196–207. Haifa, Israel, Nov 16-18. 5, 36
- [Avizienis, 1976] Avizienis, A. (1976). Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312. 30
- [Avizienis, 1985] Avizienis, A. (1985). The N-Version Approach to Fault-Tolerant Software. *Ieee Transactions on Software Engineering*, 11(12):1491–1501. 41
- [Avizienis et al., 2004] Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. 11, 12, 13, 27
- [Azambuja et al., 2010a] Azambuja, J., Sousa, F., Rosa, L., and Kastensmidt, F. (2010a). The limitations of software signature and basic block sizing in soft error fault coverage. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1 –8. 6, 47
- [Azambuja et al., 2011] Azambuja, J. R., Lapolli, A., Rosa, L., and Kastensmidt, F. L. (2011). Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique. *IEEE Transactions on Nuclear Science*, PP(99):1. On press. 48
- [Azambuja et al., 2010b] Azambuja, J. R., Souza, F., Rosa, L., and Kastensmidt, F. L. (2010b). Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and Set Faults in Microprocessors. In *11th European Conference of Radiation and Its Effects on Components and Systems. RADECS 2010*. Längenfeld, Austria. 48
- [Barth et al., 2003] Barth, J., Dyer, C., and Stassinopoulos, E. (2003). Space, atmospheric, and terrestrial radiation environments. *IEEE Transactions on Nuclear Science*, 50(3, Part 3):466–482. 3, 16

- [Baumann, 2001] Baumann, R. (2001). Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22. 16
- [Baumann, 2002] Baumann, R. (2002). Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121. 3, 18
- [Baumann, 2005a] Baumann, R. (2005a). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316. 2, 16, 24
- [Baumann, 2005b] Baumann, R. (2005b). Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266. 4, 17, 28
- [Baumann et al., 1995] Baumann, R., Hossain, T., Murata, S., and Kitagawa, H. (1995). Boron compounds as a dominant source of alpha-particles in semiconductor devices. In *1995 IEEE International Reliability Physics Proceedings, 33RD Annual*, pages 297–302. IEEE, Electron Devices Society. 4, 17, 28
- [Baumann and Smith, 2001] Baumann, R. C. and Smith, E. B. (2001). Neutron-induced 10b fission as a major source of soft errors in high density srams. *Microelectronics Reliability*, 41(2):211 – 218. 17
- [Benso et al., 2000] Benso, A., Chiusano, S., Prinetto, P., and Tagliaferri, L. (2000). A C/C++ Source-to-Source Compiler for Dependable Applications. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, DSN '00, pages 71–, Washington, DC, USA. IEEE Computer Society. 44
- [Benso et al., 2001] Benso, A., Di Carlo, S., Di Natale, G., Prinetto, P., and Tagliaferri, L. (2001). Control-flow checking via regular expressions. In *10th Asian Test Symposium, 2001. Proceedings*, pages 299 –303. 40
- [Benso and Prinetto, 2003] Benso, A. and Prinetto, P. (2003). *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Frontiers in Electronic Testing. Kluwer Academic Publishers. 51
- [Benso et al., 1998] Benso, A., Rebaudengo, M., Reorda, M. S., and Civera, P. L. (1998). An integrated HW and SW fault injection environment for real-time systems. *1998 Ieee International Symposium on Defect and Fault Tolerance in Vlsi Systems, Proceedings*, pages 117–122. 44
- [Bernardi et al., 2006a] Bernardi, P., Bolzani, L., Rebaudengo, M., Reorda, M., Vargas, F., and Violante, M. (2006a). A new hybrid fault detection technique for systems-on-a-chip. *IEEE Transactions on Computers*, 55(2):185–198. 48
- [Bernardi et al., 2010] Bernardi, P., Poehls, L. M. B., Grossi, M., and Reorda, M. S. (2010). A Hybrid Approach for Detection and Correction of Transient Faults in SoCs. *IEEE Transactions on Dependable and Secure Computing*, 7(4):439–445. 6, 48

- [Bernardi et al., 2006b] Bernardi, P., Sterpone, L., Violante, M., and Portela-Garcia, M. (2006b). Hybrid Fault Detection Technique: A Case Study on Virtex-II Pro's PowerPC 405. *IEEE Transactions on Nuclear Science*, 53(6):3550 –3557. 48
- [Berriardi et al., 2007] Berriardi, P., Bolzani, L., and Reorda, M. S. (2007). A hybrid approach to fault detection and correction in SoCs. *13th IEEE International On-Line Testing Symposium Proceedings*, pages 107–112. 48
- [Bleyer-Kocik P., 2011] Bleyer-Kocik P. (2011). PacoBlaze - A synthesizable behavioral Verilog PicoBlaze clone. <http://bleyer.org/pacoblaze>. Internet. 112
- [Bolchini, 2003] Bolchini, C. (2003). A software methodology for detecting hardware faults in VLIW data paths. *IEEE Transactions on Reliability*, 52(4):458 – 468. 46
- [Bolchini and Salice, 2001] Bolchini, C. and Salice, F. (2001). A software methodology for detecting hardware faults in VLIW data paths. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings*, pages 170 –175. 46
- [Buchner et al., 2002] Buchner, S., Marshall, P., Kniffin, S., and LaBel, K. (2002). Proton Test Guideline Development - Lessons Learned. Technical report, NASA/Goddard Space Flight Center. 53
- [Calin et al., 1996] Calin, T., Nicolaidis, M., and Velazco, R. (1996). Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, 43(6):2874–2878. 29, 35
- [Carreira et al., 1998] Carreira, J., Madeira, H., and Silva, J. (1998). Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125 –136. 54
- [Cassel and Lima, 2006] Cassel, M. and Lima, F. (2006). Evaluating one-hot encoding finite state machines for seu reliability in sram-based fpgas. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, page 6 pp. 51
- [Chapman, 2003] Chapman, K. (2003). *PicoBlaze KCPSM3. 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II Pro*. Xilinx Ltd. October. 106
- [Cheynet et al., 2000] Cheynet, P., Nicolescu, B., Velazco, R., Rebaudengo, M., Reorda, M., and Violante, M. (2000). Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6, Part 3):2231–2236. IEEE Nuclear and Space Radiation Effects Conference (NSREC), Reno, Nevada, Jul 24-28, 2000. 33, 44, 48

- [Civera et al., 2001] Civera, P., Macchiarulo, L., Rebaudengo, M., Reorda, M., and Violante, M. (2001). Exploiting circuit emulation for fast hardness evaluation. *IEEE Transactions on Nuclear Science*, 48(6):2210–2216. 55
- [Colinge, 2001] Colinge, J. P. (2001). Silicon-on-insulator technology: Overview and device physics. In *Proc. 2001 IEEE Nuclear and Space Radiation Effects Conf. Short Course*, pages 1–70. 28
- [Cuenca-Asensi et al., 2011a] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F., Guzmán-Miranda, H., and Aguirre, M. (2011a). A novel co-design approach for soft errors mitigation in embedded systems. *IEEE Transactions on Nuclear Science*, 58(3):1059–1065. 155, 161
- [Cuenca-Asensi et al., 2011b] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H., and Aguirre, M. A. (2011b). Soft core based embedded systems in critical aerospace applications. *Journal of Systems Architecture*, In Press, Corrected Proof:–. 144, 161
- [De Micheli et al., 2002] De Micheli, G., Ernst, R., and Wolf, W., editors (2002). *Readings in hardware/software co-design*. Kluwer Academic Publishers, Norwell, MA, USA. 62
- [De Micheli and Gupta, 1997] De Micheli, G. and Gupta, R. (1997). Hardware/software co-design. *Proc. of the IEEE*, 85(3):349–365. 61
- [Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197. 163
- [Del Corso et al., 2007] Del Corso, D., Passerone, C., Reyneri, L. M., Sansoè, C., Borri, M., Speretta, S., and Tranchero, M. (2007). Architecture of a small low-cost satellite. In *DSD 2007: 10TH EUROMICRO Conf. Digital System Design Architectures, Methods and Tools*, pages 428–431. Drager. 3
- [DoD, 1994] DoD (1994). MIL-HDBK-817, Military Handbook System Develop Radiation Hardness Assurance. Technical report, Department of Defense. USA. 4
- [Dodd and Massengill, 2003] Dodd, P. and Massengill, L. (2003). Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583 – 602. 18
- [Donohoe et al., 2007] Donohoe, G., Buehler, D., Hass, K., Walker, W., and Yeh, P. S. (2007). Field Programmable Processor Array: Reconfigurable Computing for Space. In *IEEE Aerospace Conference*. Big Sky, MT. 4
- [Edwards et al., 2004] Edwards, R., Dyer, C., and Normand, E. (2004). Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics. In *IEEE Radiation Effects Data Workshop (REDW)*, pages 1–5. IEEE. 3, 18

- [EEMBC, 2011] EEMBC (2011). EEMBC – The Embedded Microprocessor Benchmark Consortium. <http://www.eembc.org>. Internet. 121
- [Entrena et al., 2011] Entrena, L., López-Ongil, C., García-Valderas, M., Portela-García, M., and Nicolaidis, M. (2011). *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*, chapter 6. Hardware Fault Injection. Springer, 1st edition. 53
- [Ergin et al., 2009] Ergin, O., Unsal, O. S., Vera, X., and Gonzalez, A. (2009). Reducing Soft Errors through Operand Width Aware Policies. *IEEE Transactions on Dependable and Secure Computing*, 6(3):217–230. 5, 38
- [Ernst, 1998] Ernst, R. (1998). Codesign of embedded systems: status and trends. *Design Test of Computers, IEEE*, 15(2):45 –54. 62
- [ESA, 1993] ESA (1993). The Radiation Design Handbook ESA PSS-01-609. Technical report, European Space Agency. 4
- [ESA, 2002] ESA (2002). Single Event Effects Test Method and Guidelines. ESCC Basic Specification 25100, European Space Agency. 53
- [Fidalgo et al., 2006] Fidalgo, A. V., Alves, G. R., and Ferreira, J. M. (2006). Real time fault injection using enhanced ocd – a performance analysis. In *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06.*, pages 254 –264. 54
- [FSF, 2011] FSF (2011). GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). <http://gcc.gnu.org>. Internet. 80
- [Gaisler, 2011] Gaisler (2011). Aeroflex Gaisler Research. Página web de gaisler research. www.gaisler.com. Internet. 163
- [Geffroy and Motet, 2002] Geffroy, J.-C. and Motet, G. (2002). *Design of dependable computing systems*. Kluwer Academic Publishers, Hingham, MA, USA. 11
- [Goloubeva et al., 2005] Goloubeva, O., Rebaudengo, M., Reorda, M., and Violante, M. (2005). Improved software-based processor control-flow errors detection technique. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings*, pages 583 – 589. 41
- [Goloubeva et al., 2006a] Goloubeva, O., Rebaudengo, M., Reorda, M. S., and Violante, M. (2006a). *Software-Implemented Hardware Fault Tolerance*, volume XIV. Springer. 4, 6, 24, 26
- [Goloubeva et al., 2006b] Goloubeva, O., Rebaudengo, M., Reorda, M. S., and Violante, M. (2006b). *Software-Implemented Hardware Fault Tolerance*, volume XIV, chapter 3. Hardening the control flow. Springer. 40

- [Goloubeva et al., 2003] Goloubeva, O., Rebaudengo, M., Sonza Reorda, M., and Violante, M. (2003). Soft-error detection using control flow assertions. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings*, pages 581 – 588. 40, 48
- [Gomaa et al., 2003] Gomaa, M., Scarbrough, C., Vjaykumar, T., and Pomeranz, I. (2003). Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83. 5, 38
- [Gossett et al., 1993] Gossett, C., Hughlock, B., Katoozi, M., Larue, G., and Wender, S. (1993). Single event phenomena in atmospheric neutron environments. *IEEE Transactions on Nuclear Science*, 40(6, Part 1):1845–1852. IEEE 30th Annual International Nuclear and Space Radiation Effects Conference (NSREC), Snowbird, UT, Jul 19-23, 1993. 16
- [Gusmão-de Lima, 2000] Gusmão-de Lima, F. (2000). Single event upset mitigation techniques for programmable devices. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil. 5, 35
- [Guthaus et al., 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA. IEEE Computer Society. 121
- [Guzmán-Miranda, 2010] Guzmán-Miranda, H. (2010). *Aportaciones a las técnicas de emulación y protección de sistemas microelectrónicos complejos bajo efectos de la radiación*. PhD thesis, Escuela Superior de Ingeniería, Departamento de Ingeniería Electrónica, Universidad de Sevilla. 100
- [Guzmán-Miranda et al., 2009] Guzmán-Miranda, H., Aguirre, M., and Tombs, J. (2009). Noninvasive fault classification, robustness and recovery time measurement in microprocessor-type architectures subjected to radiation-induced errors. *IEEE Transactions on Instrumentation and Measurement*, 58(5):1514–1524. 100
- [Hentschke et al., 2002] Hentschke, R., Marques, F., Lima, F., Carro, L., Susin, A., and Reis, R. (2002). Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy. In *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design*, pages 95 – 100. 34
- [Hu et al., 2009] Hu, J., Li, F., Degalahal, V., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. (2009). Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8:27:1–27:30. 45
- [Hu et al., 2005] Hu, J. S., Li, F. H., Degalahal, V., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. (2005). Compiler-directed instruction duplication for

- soft error detection. *Design, Automation and Test in Europe Conference and Exhibition, Vols 1 and 2, Proceedings*, pages 1056–1057. 45
- [Huang et al., 2011] Huang, K., Hu, Y., and Li, X. (2011). Cross-layer optimized placement and routing for FPGA soft error mitigation. In *Design, Automation and Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6. 50
- [IBM, 2011] IBM (2011). SOI Technology: IBM's Next Advance in Chip Design. <http://www.ibm.com/>. Internet. 28
- [IEC, 2006] IEC (2006). IEC/TS 62396-1. Technical report, International Electrotechnical Commission. 4
- [IEEE, 2000] IEEE (2000). *The Authoritative Dictionary of IEEE Standards Terms*. Institute of Electrical and Electronics Engineers, Los Alamitos, CA, USA, 7th edition edition. 2, 28
- [ITRS, 2010] ITRS (2010). International Technology Roadmap for Semiconductors 2001 - 2010 Editions. Technical report, ITRS. 3, 16
- [JEDEC, 1996] JEDEC (1996). JEDEC JESD57: Test Procedures for the Measurement of Single-Event Effects in Semiconductor Devices from Heavy Ion Irradiation. Technical Report 57, JEDEC Solid State Technology Association. 53
- [JEDEC, 2006] JEDEC (2006). JEDEC Standard JESD89A. Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices. Standard 89A, JEDEC Solid State Technology Association. 53
- [JEDEC, 2009] JEDEC (2009). JEDEC - The standards resources for the world semiconductor industry. <http://www.jedec.org/>. Internet. 23
- [Jochim, 2002] Jochim, M. (2002). Detecting processor hardware faults by means of automatically generated virtual duplex systems. In *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings*, pages 399 – 408. 42
- [Johnson, 1989] Johnson, B. (1989). *Design and analysis of Fault-Tolerant Digital Systems*. Addison-Wesley. 13, 36
- [Kanawati et al., 1995] Kanawati, G., Kanawati, N., and Abraham, J. (1995). Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248 –260. 54
- [Karnik et al., 2004] Karnik, T., Hazucha, P., and Patel, J. (2004). Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143. 2, 19

- [Kastensmidt et al., 2006] Kastensmidt, F. L., Carro, L., and Reis, R. (2006). *Fault-Tolerance Techniques for SRAM-Based FPGAs*, volume 32 of *Frontiers in Electronic Testing*. Springer. XV, 183 p., ISBN: 978-0-387-31068-8. 50, 51
- [Kastensmidt et al., 2004] Kastensmidt, F. L., Neuberger, G., Hentschke, R., Carro, L., and Reis, R. (2004). Designing fault-tolerant techniques for sram-based fpgas. *Design Test of Computers, IEEE*, 21(6):552 – 562. 51
- [Kenterlis et al., 2006] Kenterlis, P., Kranitis, N., Paschalidis, A., Gizopoulos, D., and Psarakis, M. (2006). A low-cost seu fault emulation platform for sram-based fpgas. In *12th IEEE International On-Line Testing Symposium, 2006. IOLTS 2006*, page 7 pp. 54
- [Koga and Kolasinski, 1989] Koga, R. and Kolasinski, W. (1989). Heavy-Ion induced snapback in CMOS devices. *IEEE Transactions on Nuclear Science*, 36(6, Part 1):2367–2374. 21
- [Kubalik and Kubatova, 2008] Kubalik, P. and Kubatova, H. (2008). Dependable design technique for system-on-chip. *Journal of Systems Architecture*, 54(3-4):452–464. 5, 36, 50
- [Lacoe et al., 2000] Lacoe, R., Osborn, J., Koga, R., Brown, S., and Mayer, D. (2000). Application of hardness-by-design methodology to radiation-tolerant asic technologies. *IEEE Transactions on Nuclear Science*, 47(6):2334 –2341. 28
- [Laprie et al., 1992] Laprie, J. C., Avizienis, A., and Kopetz, H., editors (1992). *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 11
- [Lee and Shrivastava, 2009a] Lee, J. and Shrivastava, A. (2009a). A Compiler Optimization to Reduce Soft Errors in Register Files. *ACM Sigplan Notices*, 44(7):41–49. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tool for Embedded Systems, Dublin, Ireland, Jun 19-20, 2009. 47, 144
- [Lee and Shrivastava, 2009b] Lee, J. and Shrivastava, A. (2009b). Compiler-Managed Register File Protection for Energy-Efficient Soft Error Reduction. In *Proceedings of the ASP-DAC 2009: 14th Asia and South Pacific Design Automation Conference*, pages 618–623. Yokohama, Japan, Jan 19-22, 2009. 47, 95
- [Lee and Shrivastava, 2010] Lee, J. and Shrivastava, A. (2010). A compiler-microarchitecture hybrid approach to soft error reduction for register files. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:1018–1027. 48
- [Li and Gaudiot, 2010] Li, X. and Gaudiot, J.-L. (2010). Tolerating Radiation-Induced Transient Faults in Modern Processors. *International Journal of Parallel Programming*, 38(2):85–116. 48

- [Liem et al., 2008] Liem, C., Gu, Y. X., and Johnson, H. (2008). A compiler-based infrastructure for software-protection. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, PLAS '08*, pages 33–44, New York, NY, USA. ACM. 47
- [Lima et al., 2003] Lima, F., Carro, L., and Reis, R. (2003). Designing fault tolerant systems into sram-based fpgas. In *Design Automation Conference, 2003. Proceedings*, pages 650 – 655. 51
- [Lima et al., 2000] Lima, F., Cota, E., Carro, L., Lubaszewski, M., Reis, R., Velasco, R., and Rezgui, S. (2000). Designing a radiation hardened 8051-like micro-controller. In *Proceedings 13th Symposium on Integrated Circuits and Systems Design, 2000*, pages 255 –260. 51
- [Lin et al., 2011] Lin, S., Kim, Y.-B., and Lombardi, F. (2011). A 11-Transistor Nanoscale CMOS Memory Cell for Hardening to Soft Errors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(5):900–904. 35
- [Lindoso et al., 2011] Lindoso, A., Entrena, L., San Millán, E., Cuenca-Asensi, S., Martínez-Álvarez, A., and Restrepo-Calle, F. (2011). A co-design approach for set mitigation in embedded systems. In *Proceedings of the 12th European Conference on Radiation and its Effects on Components and Systems RADECS 2011*. Sevilla, Spain. Sept 19 - 23, 2011. 162, 164
- [Lisboa et al., 2006] Lisboa, C. A. L., Carro, L., Reorda, M. S., and Violante, M. (2006). Online hardening of programs against SEUs and SETs. In *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06*, pages 280 –290. 44
- [Lopez-Ongil et al., 2007] Lopez-Ongil, C., Garcia-Valderas, M., Portela-Garcia, M., and Entrena, L. (2007). Autonomous fault emulation: A new fpga-based acceleration system for hardness evaluation. *IEEE Transactions on Nuclear Science*, 54(1):252 –261. 55
- [Mahmood and McCluskey, 1988] Mahmood, A. and McCluskey, E. (1988). Concurrent error-detection using watchdog processors. *IEEE Transactions on Computers*, 37(2):160–174. 5, 36
- [Martínez-Álvarez et al., 2011] Martínez-Álvarez, A., Cuenca-Asensi, S., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H., and Aguirre, M. A. (2011). Compiler-directed soft error mitigation for embedded systems. *IEEE Transactions on Dependable and Secure Computing*, In Press:–. 150, 161
- [May and Woods, 1979] May, T. and Woods, M. (1979). Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9. 16
- [Mentor, 2010] Mentor (2010). Advanced FPGA Synthesis: Precision Rad-Tolerant. Data Sheet 1028010, Mentor Graphics Corp. 50, 77

- [Michalak et al., 2005] Michalak, S., Harris, K., Hengartner, N., Takala, B., and Wender, S. (2005). Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335. 3, 18
- [Miller et al., 2004] Miller, F., Buard, N., Carriere, T., Dufayel, R., Gaillard, R., Poirot, P., Palau, J.-M., Sagnes, B., and Fouillat, P. (2004). Effects of beam spot size on the correlation between laser and heavy ion seu testing. *IEEE Transactions on Nuclear Science*, 51(6):3708 – 3715. 53
- [MISRA, 2004] MISRA (2004). *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. Motor Industry Software Reliability Association. 89
- [Mitra et al., 2005] Mitra, S., Seifert, N., Zhang, M., Shi, Q., and Kim, K. S. (2005). Robust system design with built-in soft-error resilience. *Computer*, 38:43–52. 33
- [Mukherjee et al., 2002] Mukherjee, S., Kontz, M., and Reinhardt, S. (2002). Detailed design and evaluation of Redundant Multithreading alternatives. In *29th International Symposium on Computer Architecture*, pages 99–110. AK, May 25-29. 5, 37, 48
- [Mukherjee et al., 2003] Mukherjee, S., Weaver, C., Emer, J., Reinhardt, S., and Austin, T. (2003). A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *36th International Symposium on Microarchitecture, Proceedings*, pages 29–40. San Diego, CA, Dec 03-05. 55, 56, 128
- [Napoles et al., 2007] Napoles, J., Guzman, H., Aguirre, M., Tombs, J., Munoz, F., Baena, V., Torralba, A., and Franquelo, L. (2007). Radiation environment emulation for VLSI designs A low cost platform based on xilinx FPGAs. In *IEEE International Symposium on Industrial Electronics, ISIE 2007*. 77, 99
- [Neuberger et al., 2005] Neuberger, G., de Lima Kastensmidt, F., and Reis, R. (2005). An automatic technique for optimizing reed-solomon codes to improve fault tolerance in memories. *Design Test of Computers, IEEE*, 22(1):50 – 58. 34
- [Nicolaidis, 1999] Nicolaidis, M. (1999). Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proceedings of the 17th IEEE VLSI Test Symposium, 1999*, pages 86–94. 36
- [Nicolaidis, 2003] Nicolaidis, M. (2003). Data storage method with error correction. Patent pending WO/2003/038620 PCT/FR2002/003758. 34
- [Nicolaidis, 2005] Nicolaidis, M. (2005). Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405 – 418. 4, 29, 34
- [Nicolaidis, 2011a] Nicolaidis, M. (2011a). *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*. Springer, 1st edition. 3

- [Nicolaidis, 2011b] Nicolaidis, M. (2011b). *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*, chapter 8. Circuit-Level Soft-Error Mitigation. Springer, 1st edition. 33, 35
- [Nicolaidis et al., 2003] Nicolaidis, M., Achouri, N., and Boutobza, S. (2003). Dynamic Data-bit Memory Built-In Self- Repair. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD '03, pages 588–, Washington, DC, USA. IEEE Computer Society. 34
- [Nicolaidis et al., 2008] Nicolaidis, M., Perez, R., and Alexandrescu, D. (2008). Low-cost highly-robust hardened cells using blocking feedback transistors. In *26th IEEE VLSI Test Symposium. VTS 2008*, pages 371–376. 35
- [Nicolescu et al., 2004] Nicolescu, B., Savaria, Y., and Velazco, R. (2004). Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Transactions on Nuclear Science*, 51(6):3510–3518. 6, 44
- [Normand, 1996] Normand, E. (1996). Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6, Part 1):2742–2750. 33rd Annual IEEE International Nuclear and Space Radiation Effects Conference, Indian Wells, CA, JUL 15-19, 1996. 16
- [Oh and McCluskey, 2002] Oh, N. and McCluskey, E. (2002). Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability*, 51(4):392 – 402. 42
- [Oh et al., 2002a] Oh, N., Mitra, S., and McCluskey, E. J. (2002a). (EDI)-I-4: error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199. 6, 42
- [Oh et al., 2002b] Oh, N., Shirvani, P., and McCluskey, E. (2002b). Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122. 6, 33, 41, 90, 132
- [Oh et al., 2002c] Oh, N., Shirvani, P., and McCluskey, E. (2002c). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75. 6, 45, 76, 90, 98, 132
- [Palomo et al., 2010] Palomo, F., Mogollon, J., Napoles, J., and Aguirre, M. (2010). Mixed-mode simulation of bit-flip with pulsed laser. *IEEE Transactions on Nuclear Science*, 57(4):1884–1891. 53
- [Parashar et al., 2006] Parashar, A., Gurumurthi, S., and Sivasubramaniam, A. (2006). SlicK: Slice-based locality exploitation for efficient redundant multithreading. *ACM Sigplan Notices*, 41(11):95–105. 5, 38
- [Parr, 2011] Parr, T. (2011). ANTLR - ANOther Tool for Language Recognition. <http://antlr.org>. Internet. 80, 107

- [Patel and Fung, 1982] Patel, J. and Fung, L. (1982). Concurrent Error-Detection in Alus by Recomputing with Shifted Operands. *IEEE Transactions on Computers*, 31(7):589–595. 37
- [Peng et al., 2006] Peng, J., Ma, J., Hong, B., and Yuan, C. (2006). Validation of fault tolerance mechanisms of an onboard system. In *1st International Symposium on Systems and Control in Aerospace and Astronautics, 2006. ISSCAA 2006*, pages 5 pp. –1234. 54
- [Perry et al., 2007] Perry, F., Mackey, L., Reis, G. A., Ligatti, J., August, D. I., and Walker, D. (2007). Fault-tolerant typed assembly language. *Acm Sigplan Notices*, 42(6):42–53. 2, 19
- [Pflanz, 2002] Pflanz, M. (2002). *On-line error detection and fast recover techniques for dependable embedded processors*, chapter 2. Fault Models and Fault-Behavior of Processor Structures. Springer-Verlag, Berlin, Heidelberg. 24
- [Pignol, 2010] Pignol, M. (2010). COTS-based Applications in Space Avionics. In EDDA, editor, *Proceedings of the 13th Design, Automation and Test in Europe conference, DATE'10*, page 1213. ISBN: 978-3-9810801-6-2. March, Dresden, Germany. 6, 47
- [Pouponnot, 2005] Pouponnot, A. (2005). Strategic use of SEE mitigation techniques for the development of the ESA microprocessors: Past, present, and future. In *11th IEEE International On-Line Testing Symposium*, pages 319–323. IEEE Computer Society. 20, 21, 22
- [Powell et al., 1995] Powell, D., Martins, E., Arlat, J., and Crouzet, Y. (1995). Estimators for fault tolerance coverage evaluation. *IEEE Transactions on Computers*, 44(2):261–274. 53
- [Pradhan, 1996] Pradhan, D. K. (1996). *Fault-Tolerant Computer System Design*. Prentice-Hall. ISBN 0-13-057887-8, 1996. 4, 29, 31, 32
- [Ray et al., 2001] Ray, J., Hoe, J., and Falsafi, B. (2001). Dual use of superscalar datapath for transient-fault detection and recovery. In *34th Annual International Symposium on Microarchitecture (MICRO-34)*, pages 214–224. IEEE; ACM SIGMICRO, IEEE Computer Society. Austin, TX, DEC 01-05, 2001. 36
- [Rebaudengo et al., 2004] Rebaudengo, M., Reorda, M., and Violante, M. (2004). A new approach to software-implemented fault tolerance. *Journal of Electronic Testing-Theory and Applications*, 20(4):433–437. 6, 45, 48
- [Rebaudengo et al., 2002a] Rebaudengo, M., Reorda, M., Violante, M., Nicolecu, B., and Velazco, R. (2002a). Coping with seus/sets in microprocessors by means of low-cost solutions: a comparison study. *IEEE Transactions on Nuclear Science*, 49(3):1491 – 1495. 44

- [Rebaudengo et al., 1999] Rebaudengo, M., Reorda, M. S., Torchiano, M., and Violante, M. (1999). Soft-error detection through software Fault-Tolerance techniques. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages pp. 210–218. 44
- [Rebaudengo et al., 2000] Rebaudengo, M., Reorda, M. S., Torchiano, M., and Violante, M. (2000). An experimental evaluation of the effectiveness of automatic rule-based transformations for safety-critical applications. *Ieee International Symposium on Defect and Fault Tolerance in Vlsi Systems, Proceedings*, pages 257–265. 44
- [Rebaudengo et al., 2002b] Rebaudengo, M., Reorda, M. S., and Violante, M. (2002b). A new approach to software-implemented fault tolerance. In *LATW2002: IEEE Latin American Test Workshop*. 45
- [Rebaudengo et al., 2003] Rebaudengo, M., Reorda, M. S., and Violante, M. (2003). A new software-based technique for low-cost Fault-Tolerant application. *Annual Reliability and Maintainability Symposium, 2003 Proceedings*, pages 25–28. 6, 45, 76
- [Rebaudengo et al., 2001] Rebaudengo, M., Reorda, M. S., Violante, M., and Torchiano, M. (2001). A source-to-source compiler for generating dependable software. *First IEEE International Workshop on Source Code Analysis and Manipulation, Proceedings*, pages 33–42. 6, 27, 44, 76, 132
- [Rebaudengo et al., 2011] Rebaudengo, M., Sonza-Reorda, M., and Violante, M. (2011). *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*, chapter 9. Software-Level Soft Error Mitigation Techniques. Springer, 1st edition. XVIII, 368 p. 59 illus., Hardcover. ISBN: 978-1-4419-6992-7. 38
- [Reddy et al., 2006] Reddy, V. K., Parthasarathy, S., and Rotenberg, E. (2006). Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. *ACM Sigplan Notices*, 41(11):83–94. 5, 38
- [Reed and Solomon, 1960] Reed, I. S. and Solomon, G. (1960). Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304. 34
- [Reinhardt and Mukherjee, 2000] Reinhardt, S. and Mukherjee, S. (2000). Transient fault detection via simultaneous multithreading. In *27th Int. Symp. on Computer Architecture*, pages 25–36. Vancouver, Canada, Jun 12-14. 37, 97
- [Reis et al., 2005a] Reis, G., Chang, J., Vachharajani, N., Mukherjee, S., Rangan, R., and August, D. (2005a). Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture, Proceedings*, pages 148–159. Madison, WI, Jun 04-08. 6, 48, 56, 129

- [Reis et al., 2007] Reis, G. A., Chang, J., and August, D. I. (2007). Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47. 6, 46, 114, 117, 132
- [Reis et al., 2005b] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. (2005b). SWIFT: software implemented fault tolerance. *CGO 2005: Int Symposium on Code Generation and Optimization*, pages 243–254. 6, 45, 48, 76, 90, 98, 132
- [Reis et al., 2005c] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I., and Mukherjee, S. S. (2005c). Software-Controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, Vol. V:1–28. 48
- [Restrepo-Calle et al., 2011] Restrepo-Calle, F., Cuenca-Asensi, S., Aguirre, M. A., Palomo, F. R., Guzmán-Miranda, H., and Martínez-Álvarez, A. (2011). On the definition of real conditions for a fault injection experiment on embedded systems. In *Proceedings of the 12th European Conference on Radiation and its Effects on Components and Systems RADECS 2011*. Sevilla, Spain. Sept 19 - 23, 2011. 161
- [Restrepo-Calle et al., 2010a] Restrepo-Calle, F., Martínez-Álvarez, A., Cuenca-Asensi, S., Palomo, F., and Aguirre, M. (2010a). Hardening development environment for embedded systems. In the 2nd HiPEAC Workshop on Design for Reliability (DFR'10) held in conjunction with The 5th Int. Conf. on High Performance and Embedded Architectures and Compilers. Pisa, Italy, Jan 25-27, 2010. 162
- [Restrepo-Calle et al., 2010b] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., Aguirre, M., and Cuenca-Asensi, S. (2010b). A compiler-based infrastructure for fault-tolerant co-design. In *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems, SCOPES*. Art. no. 4, St. Goar, Germany, 28-29 June 2010. 162
- [Restrepo-Calle et al., 2010c] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., Aguirre, M., and Cuenca-Asensi, S. (2010c). Mitigación automática de soft-errors en nuevas aplicaciones de los sistemas empotrados. In *Actas del I Simposio en Computación Empotrada, SiCE 2010*, pages 25–32. Valencia, España, Septiembre 2010. ISBN: 978-84-92812-69-1. 163
- [Restrepo-Calle et al., 2010d] Restrepo-Calle, F., Martínez-Álvarez, A., Guzmán-Miranda, H., Palomo, F., and Cuenca-Asensi, S. (2010d). Application-driven co-design of fault-tolerant industrial systems. In *Proceedings of the IEEE International Symposium on Industrial Electronics, ISIE*. Bari, Italy. July 4-7. 2010. 162

- [Restrepo-Calle et al., 2010e] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F., Guzmán-Miranda, H., Aguirre, M., and Cuenca-Asensi, S. (2010e). Prototipado Rápido de Sistemas Empotrados Tolerantes a Radiación en FPGA. In *Actas de las X Jornadas de Computación Reconfigurable y Aplicaciones JCRA 2010*, pages 261–268. Valencia, España, Septiembre 2010. ISBN: 978-84-92812-56-1. 162
- [Restrepo-Calle et al., 2010f] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F., Guzmán-Miranda, H., Aguirre, M., and Cuenca-Asensi, S. (2010f). Rapid Prototyping of Radiation-Tolerant Embedded Systems on FPGA. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications FPL 2010*, pages 326 – 331. Milano, Italy. Aug 31 - Sep 2. 2010. 162
- [Restrepo-Calle et al., 2010g] Restrepo-Calle, F., Martínez-Álvarez, A., Palomo, F. R., Guzmán-Miranda, H., Aguirre, M. A., and Cuenca-Asensi, S. (2010g). A novel co-design approach for soft errors mitigation in embedded systems. In *Proceedings of the 11th European Conference on Radiation and its Effects on Components and Systems RADECS 2010*. Langenfeld, Austria. Sept 20 - 24, 2010. 162
- [Roche et al., 2004] Roche, P., Jacquet, F., Caillat, C., and Schoellkopf, J.-P. (2004). An alpha immune and ultra low neutron ser high density sram. In *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 671 – 672. 4, 28
- [Rotenberg, 1999] Rotenberg, E. (1999). AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, pages 84–91. IEEE Computer Society. Madison, WI, JUN 15-18, 1999. 37
- [RTCA, 2000] RTCA (2000). DO-254 Design Assurance Guideline for Airborne Electronic Hardware. Technical report, RTCA Inc. 4
- [Saihalasz et al., 1982] Saihalasz, G., Wordeman, M., and Dennard, R. (1982). Alpha-particle-induced soft error in VLSI circuits. *IEEE Transactions on Electron Devices*, 29(4):725–731. 16
- [Samudrala et al., 2004] Samudrala, P., Ramos, J., and Katkoori, S. (2004). Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. *IEEE Transactions on Nuclear Science*, 51(5, Part 4):2957–2969. 5, 38
- [Saxena and McCluskey, 1998] Saxena, N. and McCluskey, E. (1998). Dependable adaptive computing systems - The ROAR project. In *1998 IEEE International Conference on Systems, Man, and Cybernetics, VOLS 1-5*, pages 2172–2177. San Diego, CA, Oct 11-14, 1998. 37
- [Scholzel, 2010] Scholzel, M. (2010). HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In

- Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 723–728. 48
- [Schrimpf, 2007] Schrimpf, R. D. (2007). Radiation effects in microelectronics. In Velazco, R and Fouillat, P and Reis, R, editor, *Radiation Effects on Embedded Systems*, pages 11–29. Springer. 17, 20
- [Sexton et al., 1997] Sexton, F., Fleetwood, D., Shaneyfelt, M., Dodd, P., and Hash, G. (1997). Single event gate rupture in thin gate oxides. *IEEE Transactions on Nuclear Science*, 44(6, Part 1):2345–2352. 22
- [Shirvani et al., 2000] Shirvani, P., Saxena, N., and McCluskey, E. (2000). Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3):273–284. 47
- [Shivakumar et al., 2002] Shivakumar, P., Kistler, M., Keckler, S., Burger, D., and Alvisi, L. (2002). Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks, Proceedings*, pages 389–398. 2, 16
- [Srour et al., 2003] Srour, J., Marshall, C., and Marshall, P. (2003). Review of displacement damage effects in silicon devices. *IEEE Transactions on Nuclear Science*, 50(3, Part 3):653–670. 20
- [Stassinopoulos et al., 1992] Stassinopoulos, E., Brucker, G., Calvel, P., Baiget, A., Peyrotte, C., and Gaillard, R. (1992). Charge generation by heavy-ions in power MOSFETS, burnout space predictions, and dynamic SEB sensitivity. *IEEE Transactions on Nuclear Science*, 39(6, Part 1):1704–1711. 22
- [Serpone et al., 2007] Serpone, L., Sonza Reorda, M., Violante, M., Kastensmidt, F. L., and Carro, L. (2007). Evaluating Different Solutions to Design Fault Tolerant Systems with SRAM-based FPGAs. *Journal of Electronic Testing: Theory and Applications*, 23:47–54. 51
- [Serpone and Violante, 2006] Serpone, L. and Violante, M. (2006). A new reliability-oriented place and route algorithm for SRAM-based FPGAs. *IEEE Transactions on Computers*, 55(6):732–744. 50
- [Storey, 1996] Storey, N. (1996). *Safety-Critical Computer Systems*. Addison Wesley, 2 edition edition. ISBN 0-201-42787-7. 13
- [Storey and Maly, 1990] Storey, T. and Maly, W. (1990). CMOS Bridging Fault-Detection. In *Proceedings: 21st International Test Conference 1990 - The Changing Philosophy of Test*, pages 842–851. IEEE, Computer Society Press. 25
- [Synopsys, 2010] Synopsys (2010). Synopsys FPGA Synthesis Synplify Pro Reference Manual. Technical report, Synopsys Inc. Actel Edition. 50, 77

- [Taber, A. H. and Normand, E., 1995] Taber, A. H. and Normand, E. (1995). Investigation and characterization of seu effects and hardening strategies in avionics. Technical Report 92, Defense Nuclear Agency. IBM Report 92-L75-020-2. 3, 17
- [TexasInstruments, 2011] TexasInstruments (2011). MSP340 Microcontroller - 16-bit ultra low power microcontroller. http://www.ti.com/lscds/ti/microcontroller/16-bit_msp430/overview.page. Internet. 163
- [Velazco et al., 2000] Velazco, R., Rezgui, S., and Ecoffet, R. (2000). Predicting error rate for microprocessor-based digital architectures through c.e.u. (code emulating upsets) injection. *IEEE Transactions on Nuclear Science*, 47(6):2405–2411. 54
- [Vemu and Abraham, 2006] Vemu, R. and Abraham, J. (2006). CEDA: control-flow error detection through assertions. In *12th IEEE International On-Line Testing Symposium, 2006. IOLTS 2006*, page 6 pp. 41
- [Vemu and Abraham, 2008] Vemu, R. and Abraham, J. (2008). Budget-dependent control-flow error detection. In *14th IEEE International On-Line Testing Symposium, 2008. IOLTS '08*, pages 73–78. 41
- [Venkatasubramanian et al., 2003] Venkatasubramanian, R., Hayes, J., and Murray, B. (2003). Low-cost on-line fault detection using control flow assertions. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003*, pages 137 – 143. 40
- [Vierhaus et al., 1993] Vierhaus, H., Meyer, W., and Glaser, U. (1993). CMOS Bridges and Resistive Transistor Faults - IDDQ versus Delay Effects. In *International Test Conference 1993 PROCEEDINGS - Designing, Testing, and Diagnostics - Join Them*, pages 83–91. IEEE, Computer Society, IEEE. Baltimore, MD, Oct 17-21, 1993. 25
- [Vijaykumar et al., 2002] Vijaykumar, T., Pomeranz, I., and Cheng, K. (2002). Transient-fault recovery using simultaneous multithreading. In *29th Annual International Symposium on Computer Architecture*, pages 87–98. Anchorage, AK, May 25-29, 2002. 5, 37
- [Von-Neumann, 1956] Von-Neumann, J. (1956). Probabilistic logics and synthesis of reliable organisms from unreliable components. In Shannon, C. E. and J. McCarthy, E., editors, *Automata Studies*, pages 43–98, Princeton, NJ. Princeton Univ. 5, 29, 36, 114
- [Wang and Agrawal, 2008] Wang, F. and Agrawal, V. (2008). Single Event Upset: An Embedded Tutorial. In *21st International Conference on VLSI Design, 2008. VLSID 2008*, pages 429 –434. 16
- [Wolf, 1994] Wolf, W. (1994). Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967 –989. 61

- [Xilinx, 2000] Xilinx (2000). Virtex Series Configuration Architecture User Guide. Application Notes 151, Xilinx Inc. 51
- [XILINX, 2008] XILINX (2008). *PicoBlaze 8-bit Embedded Microcontroller User Guide. UG129 (v1.1.2)*. Xilinx Ltd. June 2008. 105, 112
- [Xilinx, 2009] Xilinx (2009). Aerospace and defense: Xilinx TMRtool. Technical Report , Xilinx Inc. 50, 77
- [Xilinx, 2010] Xilinx (2010). Radiation-Hardened, Space-Grade Virtex-5QV FPGA Data Sheet: DC and Switching Characteristics. Data Sheet DS692 (v1.0.1), Xilinx Inc. 50
- [Xilinx, 2011] Xilinx (2011). FPGA, CPLD, and EPP Solutions from Xilinx, Inc. <http://www.xilinx.com/>. Internet. 29
- [Yau and Chen, 1980] Yau, S. and Chen, F. (1980). An approach to concurrent control flow checking. *IEEE Transactions on software engineering*, 6(2):126–137. 90
- [Ziegler and Lanford, 1981] Ziegler, J. and Lanford, W. (1981). The effect of sea-level cosmic-rays on electronic devices. *Journal of applied physics*, 52(6):4305–4312. 16
- [Ziegler and Puchner, 2004] Ziegler, J. and Puchner, H. (2004). *SER — History, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corp. 3, 18
- [Zoellin et al., 2008] Zoellin, C. G., Wunderlich, H.-J., Polian, I., and Becker, B. (2008). Selective hardening in early design steps. In *Proceedings of the 2008 13th European Test Symposium*, pages 185–190, Washington, DC, USA. IEEE Computer Society. 38



Universitat d'Alacant
Universidad de Alicante

Lista de acrónimos

ACE *Architecturally Correct Execution*

ACFC *Assertions for Control Flow Checking*

ALU *Arithmetic Logic Unit*

ARB_T *Automatic Ruled Based Transformation*

ARB_T-FT *Automatic Ruled Based Transformation for Fault Tolerance*

API *Application Programming Interface*

ASIC *Application-Specific Integrated Circuit*

AST *Abstract Syntax Tree*

AVF *Architectural Vulnerability Factor*

BISR *Built-In Self-Repair*

BPSG *Borophosphosilicate glass*

CAM *Content-Addressable Memories*

CEDA *Control-flow Error Detection through Assertions*

CFCSS *Control-Flow Checking by Software Signatures*

CFE *Control Flow Error*

CFG *Control Flow Graph*

CMOS *Complementary Metal-Oxide-Semiconductor*

COTS *Commercial Off The Shelf*

CRAFT *CompileR Assisted Fault Tolerance*

CRC *Cyclic Redundancy Check*

CRT *Chip-level Redundantly Threaded*

CRTR *Chip-level Redundant Threading with Recovery*

DD *Displacement Damage*

DICE *Dual Interlocked Cell*

DIVA *Dynamic Implementation Verification Architecture*

DMR *Dual Modular Redundancy*

ECC *Error Correction Code*

ED⁴I *Error Detection by Diverse Data and Duplicated Instructions*

EDAC *Error Detection and Correction Code*

EDC *Error Detection Code*

EDDI *Error Detection by Duplicated Instructions*

FIR *Finite Impulse Response*

FPGA *Field Programmable Gate Array*

GIF *Generic Instruction Flow*

HDL *Hardware Description Language*

HGIF *Hardened Generic Instruction Flow*

ILP *Instruction Level Parallelism*

ISA *Instruction Set Architecture*

ISS *Instruction Set Simulator*

LRC *Longitudinal Redundancy Check*

MBU *Multiple Bit Upset*

MCU *Multiple Cell Upset*

MOS *Metal-Oxide-Semiconductor*

MOSFET *Metal-Oxide-Semiconductor Field Effect Transistor*

MTTF *Mean Time To Failure*

MUT *Module Under Test*

MWTF *Mean Work To Failure*

PID *Proportional Integral Derivative*

RAM *Random-Access Memory*

REDWC *REcomputing with Duplication with Comparison*

RESO *REcomputing with Shifted Operands*

RESWO *REcomputing with Swapped Operands*

RMT *Redundant Multi-Threading*

RTL *Register Transfer Level*

SDC *Silent Data Corruption*

SEB *Single Event Burnout*

SEE *Single Event Effect*

SEFI *Single Event Functional Interrupt*

SEGR *Single Event Gate Rupture*

SEL *Single Event Latch-up*

SEMU *Single Event Multiple Upsets*

SES *Single Event Snapback*

SET *Single Event Transient*

SEU *Single Event Upset*

SHE *Software Hardening Environment*

SMT *Simultaneous Multi-Threading*

SPCD *Selective Procedure Call Duplication*

SRAM *Static Random Access Memory*

SRTR *Simultaneous and Redundant Multi-Threaded processor with Recovery*

SoC *System on Chip*

SOI *Silicon on Insulator*

SoR *Sphere of Replication*

SWIFT *SoftWare Implemented Fault Tolerance*

SWIFT-R *SoftWare Implemented Fault Tolerance Recovery*

TID *Total Ionizing Dose*

TMR *Triple Modular Redundancy*

UML *Unified Modeling Language*

unACE *unnecessary for Architecturally Correct Execution*

VLIW *Very Long Instruction Word*

YACCA *Yet Another Control flow Checking Approach*

