Universidad Carlos III de Madrid
www.uc3m.es

TESIS DOCTORAL

Técnicas híbridas de tolerancia a fallos en microprocesadores

Autor:

Luis Isaías Parra Avellaneda

Directores:

Dr. Luis Alfonso Entrena Arrontes

Dra. Almudena Lindoso Muñoz

DEPARTAMENTO DE TECNOLOGÍA ELECTRÓNICA

Leganés, Julio 2017

Universidad Carlos III de Madrid
www.uc3m.es

TESIS DOCTORAL

TÉCNICAS HÍBRIDAS DE TOLERANCIA A FALLOS EN MICROPROCESADORES

Autor: Luis Isaías Parra Avellaneda

Directores: Dr. Luis Alfonso Entrena Arrontes

Dra. Almudena Lindoso Muñoz

Firma del Tribunal Calificador:

Firma

Presidente: (Nombre y apellidos)

Vocal: (Nombre y apellidos)

Secretario: (Nombre y apellidos)

Calificación:

Leganés/Getafe, de de

Quiero dedicar este trabajo a mi familia

A mis padres

A Laura, a Aurora y a Luis

Agradecimientos

En primer lugar me gustaría mostrar mi agradecimiento a cada uno de los miembros del grupo de Diseño Microelectrónico y Aplicaciones, dentro del cual he realizado esta tesis doctoral. Gracias a Luis Entrena y a Almudena Lindoso por vuestra labor como directores de tesis, por vuestro continuo apoyo tanto en el ámbito personal como en el profesional. Gracias a Celia López, a Marta Portela y a Mario García por vuestra inestimable ayuda y por todo lo que he aprendido de vosotros a lo largo de estos años.

Gracias a mis compañeros del Departamento de Tecnología Electrónica, con los que he compartido tan buenos momentos a lo largo de mi estancia en la universidad.

Por último, quería agradecer a mi familia el continuo apoyo y ánimo que me han mostrado a lo largo de la realización de la tesis.

Resumen de la Tesis

Este trabajo de tesis doctoral presenta tres nuevas técnicas de detección de errores de control de flujo en microprocesadores. Estas técnicas han sido implementadas de manera no intrusiva en un módulo hardware externo y se han combinado con técnicas de software para obtener una elevada capacidad de detección de errores. Se ha utilizado la interfaz de traza como medio de observación de la ejecución permitiendo la detección de errores de flujo de una manera no intrusiva y sin penalización en las prestaciones.

La primera técnica propuesta ha sido denominada Predicción del Contador de Programa. Está técnica está basada en el cálculo del contador de programa a partir del código de instrucción actual y el valor previo del contador de programa. Esta técnica es capaz de detectar el subconjunto de errores que afectan al contador de programa de una manera muy eficiente y con un coste en términos de recursos necesarios para su implementación muy reducido. Adicionalmente, es importante destacar que la técnica Predicción del Contador de Programa es complementaria a las otras dos técnicas descritas en la tesis, Monitorización de firmas y Monitorización dual, y es necesaria para que éstas dos alcancen elevadas tasas de detección de errores de control de flujo.

La técnica de Monitorización de Firmas realiza el cálculo de una firma online con el objetivo de verificar la ejecución de un bloque de instrucciones. Cada bloque tiene asignada una firma de referencia que es calculada en tiempo de compilación. La firma de referencia y la calculada en tiempo de ejecución son comparadas al final de la ejecución de cada bloque. Una de las mejoras que presenta esta técnica respecto a técnicas basadas en el cálculo de firmas propuestas por otros autores es que consigue reducir el tamaño de la memoria necesaria para almacenar las firmas de referencia utilizando una tabla denominada CFC-ST. Se han propuesto dos métodos de almacenamiento y acceso a la tabla CFC-ST, un método estático y un método dinámico. En ambos casos el impacto sobre

el rendimiento del sistema es reducido al conseguir reducir el tamaño de la tabla necesaria para almacenar las firmas de referencia.

La tercera técnica desarrollada en esta tesis doctoral es la técnica de Monitorización Dual. Esta técnica monitoriza la ejecución utilizando dos puntos de observación diferentes, la interfaz de traza y el bus de memoria. Gracias a estos dos puntos de observación se obtiene información de la ejecución en diferentes etapas del pipeline del microprocesador. En concreto se realiza la comparación del contador de programa y el código de instrucción obtenidos en la etapa de búsqueda con el valor de los mismos justo después de la etapa de ejecución.

Uno de los puntos a destacar de las técnicas Monitorización Dual y Predicción del Contador de Programa es que pueden ser implementadas sin necesidad de almacenar información de la ejecución calculada en tiempo de compilación, reduciendo de una manera considerable el impacto sobre el rendimiento del sistema, ya que el número de recursos necesarios para su implementación es muy reducido.

Se han realizado extensas campañas de inyección de fallos utilizando la herramienta AMUSE, lo cual nos ha permitido evaluar la efectividad de las tres técnicas propuestas en esta tesis doctoral. Los resultados obtenidos en cada una de las campañas de inyección demuestran que las técnicas presentadas en esta tesis detectan de una manera eficiente aquellos errores que afectan al flujo de programa alcanzando una buena solución de compromiso entre la cobertura a fallos y el impacto sobre el rendimiento del sistema.

ÍNDICE

LISTA DE ACRÓNIMOS

ALU	***Arithmetic Logic Unit***
AHB	***AMBA High-performance Bus***
AMBA	***Advanced Microcontroller Bus***
AMUSE	***Autonomous MUltilevel emulation system for Soft Error evaluation***
APB	***Advanced Peripheral Bus***
AR-SMT	***Active-Stream/Redundant-Stream Simultaneous multithreading***
ASIS	***Asynchronous SIS***
BB	***Basic Block***
BID	***Branch free Interval Identifier***
CEDA	***Control-Flow Error Detection Using Assertions***
CFC	***Control Flow Checking***
CFC-ST	***CFC Signature Table***
CFCSS	***Control Flow Checking by Software Signature***
CFE	***Control flow error***
CFID	***Control Flow Identifier***
CMS	***Continuous Signature Monitoring***
COTS	***Commercial Off-The-Shelf***
DD	***Data Duplication***
ECCA	***Enhanced High level control flow checking approach using assertions***
EDDI	***Error-Detection by Duplicated Instructions***

FIFO *First Input First Output*

FPGA *Field Programmable Gate Array*

FV *Final Variables*

GPIO *General Purpose Input Output*

HM *Hardware Monitor*

IP *Intelectual Property*

IR *Instruction Register*

IV *Inverted Branches + Variables*

LFSR *Linear Feedback Shift Register*

MBU *Multiple Bit Upset*

MISR *Multiple Input Signature Register*

OCFCM *Online control-flow checker module*

OSLC *On-line Signature Learning and Checking*

PC *Program Counter*

PSA *Path Signature Analysis*

RAM *Read Access Memory*

RISC *Reduced Instruction Set Computer*

SBU *Single Bit Upset*

SDC *Silent Data Corruption*

SEE *Single Event Effect*

SEFI *Single-event functional interrupt*

SEL *Single-event latchup*

SET *Single Event Transient*

SEU *Single Event Upset*

SHE *Software Hardening Environment*

SIC *Structural Integrity Checking*

SIS *Signature Instruction Streams*

SMT *Simultaneous multithreading*

SoC *System on Chip*

SPARC *Scalable Processor Architecture*

SPCD *Selective Procedure Call Duplication*

TMR *Triple Modular Redundancy*

UART *Universal Asynchronous receiver transmitter*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very High Speed Integrated Circuit*

YACCA *Yet Another Control Flow Checking Approach*

LISTA DE FIGURAS

LISTA DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN

1. Introducción

Este trabajo de tesis doctoral propone y desarrolla nuevas técnicas híbridas de detección de errores en microprocesadores. El trabajo realizado está orientado a la búsqueda de nuevas soluciones de tolerancia a fallos que mejoren el compromiso entre la cobertura de fallos y el impacto sobre el área y las prestaciones de los sistemas basados en microprocesador.

En este primer capítulo se presenta una breve introducción a la problemática tratada en la tesis doctoral, planteando el problema que se quiere resolver así como los objetivos perseguidos. En el primer apartado se justifica la importancia de la detección de errores en sistemas basados en microprocesador, así como la necesidad de buscar soluciones de compromiso entre la tolerancia a fallos y el impacto sobre el rendimiento del sistema. En el segundo apartado se establecen los objetivos que se pretenden alcanzar en este trabajo de tesis doctoral. Finalmente, en el tercer apartado se muestra la estructura del resto del documento.

1.1. Motivación

Los microprocesadores tienen una presencia abrumadora en infinidad de aplicaciones esenciales para nuestra sociedad y todo apunta a que el equipamiento de computación va a seguir creciendo en cantidad y en complejidad. Además, la tendencia actual hacia sistemas cada vez más "inteligentes" y con mayor autonomía contribuye a que la repercusión potencial de los fallos sea mayor y por tanto a que las necesidades de fiabilidad aumenten. Por tanto, asegurar un correcto funcionamiento durante la ejecución de un sistema basado en microprocesador es un problema de gran relevancia actual.

Los circuitos digitales son susceptibles de sufrir fallos como consecuencia del impacto de partículas de alta energía, principalmente neutrones a nivel terrestre o iones de diferentes tipos en el entorno espacial, los cuales pueden producir perturbaciones transitorias en los nodos del circuito. Este tipo de fallos se conocen como *Single Event Effects* (SEEs). Tradicionalmente los SEEs

han supuesto un problema y un aspecto a tener en cuenta a la hora de diseñar un sistema en entornos dónde la radiación es importante, como por ejemplo en aplicaciones aeroespaciales. Sin embargo, debido a la evolución de la tecnología y a la elevada complejidad de los circuitos digitales actuales, el fenómeno de los SEEs se ha extendido a entornos de aplicación terrestre dónde los niveles de radiación son mucho menores.

Los SEEs pueden producir fallos permanentes o transitorios. Estos últimos, que se conocen generalmente como *soft errors*, no dañan el circuito físicamente pero pueden alterar su funcionamiento y son cada vez más frecuentes. A su vez, los *soft errors* pueden dividirse en fallos que afectan a los elementos de memoria (biestables o memorias) y en fallos que afectan a la lógica combinacional. En este trabajo nos centraremos principalmente en dos tipos de *soft errors, denominados Single Event Upset* (*SEU*) y *Single Event Transient* (*SET*) respectivamente.

Un *SEU* se produce cuando una partícula impacta en un transistor perteneciente a un elemento de memoria con una carga suficiente como para provocar que el transistor conmute y cambie el valor lógico almacenado. Este tipo de fallo se modela invirtiendo el valor lógico almacenado en el elemento de memoria, el cual queda modificado hasta que se escribe un nuevo valor. Este modelo es ampliamente aceptado por la comunidad científica y comúnmente se le denomina *bit-flip*.

Por otro lado, un *SET* se produce cuando una partícula impacta sobre un transistor que pertenece a la lógica combinacional. El impacto de la partícula produce un pulso de tensión de corta duración, del orden de varios cientos de picosegundos [Poup05], el cual puede propagarse hasta las salidas del circuito o hasta uno o varios elementos de memoria. Este efecto además se ve agravado con el aumento de las frecuencias de funcionamiento que se están produciendo en los circuitos digitales actuales. Por tanto, un *SET* puede producir múltiples *bit-flip* en el circuito que sufre el fallo.

Como se ha comentado anteriormente, la sensibilidad de los circuitos digitales a los *soft errors* ha crecido hasta niveles inaceptables en diferentes ámbitos de aplicación, tanto en entornos con niveles de radiación elevados como en entornos terrestres dónde los niveles de radiación son considerablemente menores. Este aumento de la sensibilidad a *soft errors* se ha producido como consecuencia de la reducción de los tamaños de los transistores, las elevadas densidades de integración, la reducción de las tensiones de alimentación y la utilización de frecuencias de reloj más elevadas. Hay otros efectos, como el envejecimiento y la variabilidad, que también producen efectos relacionados en los circuitos digitales modernos.

La disminución del tamaño de los transistores supone que la zona sensible a un *SEE* es menor. Sin embargo, como consecuencia de esta disminución del tamaño de los transistores se ha producido también un aumento de las densidades de integración de los circuitos. Por tanto, el número de transistores que se utilizan en un circuito aumenta, provocando que la probabilidad de que se produzca un fallo aumente.

La disminución de la tensión de alimentación implica que la carga mínima necesaria para producir un cambio en el valor lógico de un elemento del circuito se vea reducida. Esto se traduce en que la energía necesaria para que se produzca un *soft error* es menor y, por tanto, partículas de menor energía pueden provocar un cambio en el valor de tensión de una celda, resultando en un fallo en el circuito.

El incremento de las frecuencias de funcionamiento supone un aumento en la probabilidad de que un fallo originado en la lógica combinacional sea capturado por elementos de memoria, produciendo un *bit-flip*.

Como consecuencia de esta mayor sensibilidad a los *soft errors* en los circuitos digitales, la protección on-line se ha convertido en una necesidad ineludible para cumplir los requisitos de fiabilidad. Hoy en día los microprocesadores son el corazón de la mayoría de los circuitos digitales. Los Sistemas "On-Chip" (SoC) pueden contener varios microprocesadores, memorias y otros componentes comúnmente usados en una amplia

variedad de aplicaciones, incluyendo ámbitos de fiabilidad crítica, como por ejemplo, la industria del automóvil, la biomedicina, la industria aeroespacial y las telecomunicaciones. En estos campos de aplicación, un fallo en la aplicación puede producir un resultado computacional erróneo o la pérdida de control del sistema, provocando graves consecuencias en el funcionamiento del mismo. Un fallo en uno de estos sistemas críticos puede suponer elevadas pérdidas económicas e incluso daños personales, por lo que es de vital importancia que los circuitos sean tolerantes a fallos.

Para lograr la necesaria tolerancia a fallos es preciso implementar técnicas que permitan al circuito detectar o corregir los errores que se produzcan. Este objetivo de fiabilidad debe de ser alcanzado teniendo en cuenta además los requisitos habituales en cuanto a coste, rendimiento, consumo y tiempo de puesta en el mercado.

Existen diferentes soluciones para mejorar la tolerancia a fallos que pueden ser implementadas en hardware o en software. Las soluciones hardware son generalmente las más efectivas, pero tienen el inconveniente de que modificar el hardware exige un esfuerzo de desarrollo notable que no siempre es factible. Por este motivo, en la medida de lo posible se prefieren soluciones no intrusivas, es decir, que permitan la detección de errores sin necesidad de modificar el hardware existente, generalmente mediante módulos hardware externos que observen el funcionamiento del procesador. Para este propósito, es necesario considerar las posibles interfaces a través de las cuales se puede realizar dicha observación.

El punto de observación que se ha utilizado en la mayoría de los trabajos existentes es el bus entre el microprocesador y la memoria. Desde esta interfaz se pueden obtener las direcciones y las instrucciones que entran al microprocesador. Recientemente se ha propuesto una solución alternativa basada en el uso de las infraestructuras de depuración [Port12], y en particular de la interfaz de traza. Las infraestructuras de depuración están diseñadas para facilitar la depuración del código durante la fase de desarrollo del software y son muy comunes en los procesadores modernos. Estas infraestructuras permiten el acceso a recursos internos del procesador,

tales como los contenidos de los registros. En particular, la interfaz de traza facilita la trazabilidad del código ejecutado, observando el flujo de instrucciones sin interferir en su ejecución. Puesto que estas infraestructuras no se usan durante la operación nominal del procesador, pueden reaprovecharse para la monitorización on-line. Por sus inherentes ventajas, esta es la opción que se utilizará por defecto en esta tesis.

Las técnicas de tolerancia a fallos que se utilizan en la actualidad se basan generalmente en la adición de redundancia mediante replicación total del microprocesador o de la ejecución. En consecuencia, el impacto en el área, el consumo y las prestaciones de los microprocesadores es muy grande. Por tanto, existe una necesidad de buscar soluciones más eficientes que proporcionen un mejor compromiso entre la cobertura de fallos y la penalización en el área y el rendimiento del sistema.

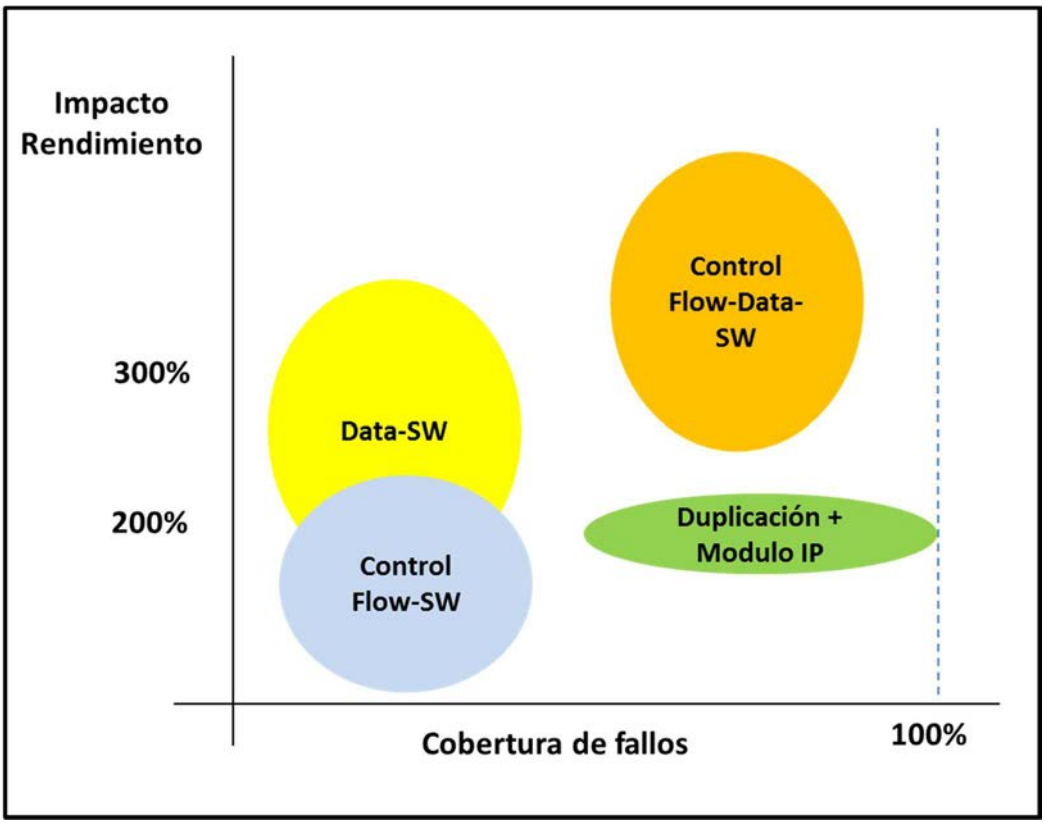

Figura 1: Cobertura a fallos e impacto de las distintas técnicas para la detección de errores.

En la Figura 1 se ilustran de manera cualitativa la cobertura a fallos y el impacto sobre el rendimiento de las técnicas existentes de tolerancia a

fallos. Habitualmente se distingue entre dos grandes tipos de errores, que requieren técnicas diferentes: errores de datos, que afectan a los datos calculados; y errores de control de flujo, que afectan al correcto orden de ejecución de las instrucciones. Las técnicas asociadas se pueden implementar mediante software, hardware o una combinación de ambas, lo que se conoce como técnicas híbridas.

Las técnicas que se basan exclusivamente en el software (*Data-SW, Control Flow-SW*) requieren grandes cantidades de instrucciones redundantes, lo que produce un impacto muy elevado en el rendimiento del sistema. Además, las coberturas de fallos son limitadas debido a que algunos fallos son muy difíciles de detectar o corregir desde la misma aplicación que los sufre. Una solución alternativa consiste en replicar completamente la ejecución, bien mediante dos procesadores realizando la misma ejecución en paralelo o bien repitiendo la ejecución en un mismo procesador, y utilizar un módulo hardware encargado de comprobar que ambas ejecuciones han sido correctas (*Duplicación + Módulo IP*). Con esta solución se pueden alcanzar coberturas de fallos cercanas al 100% [Port12] con un impacto sobre el rendimiento del sistema en torno al 200%. Esta aproximación es la que se ha tomado como punto de referencia para desarrollar nuevas técnicas que sean más eficientes.

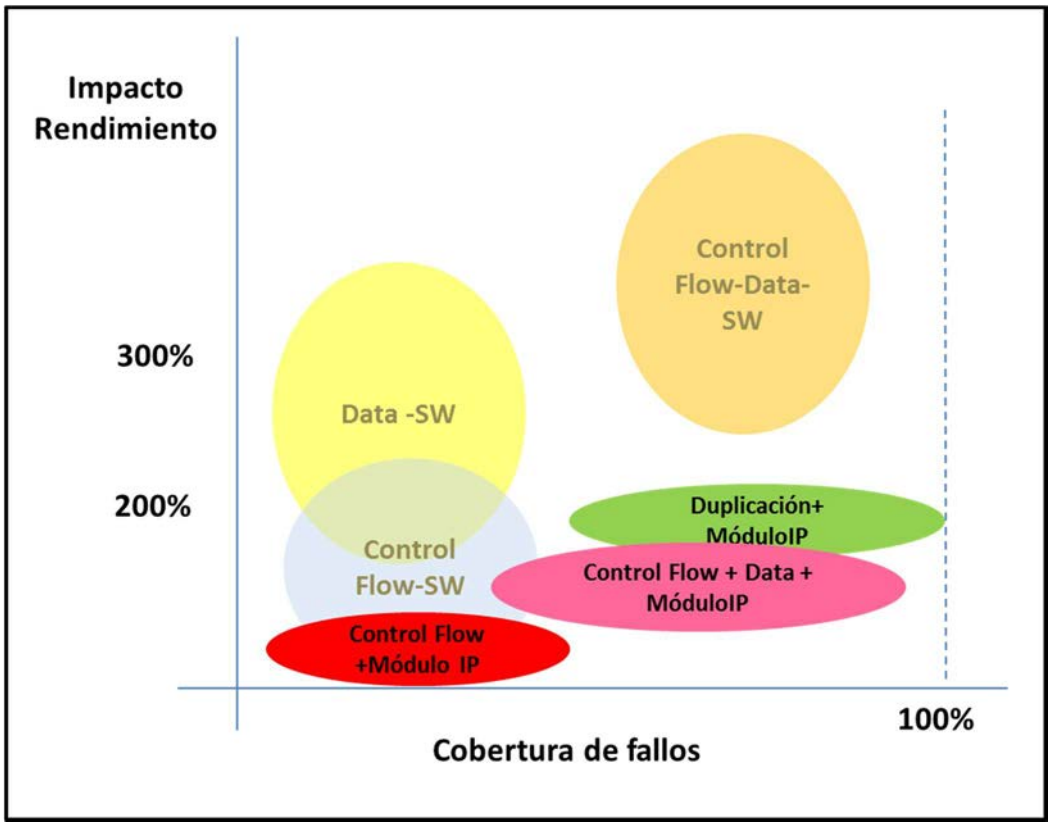

Figura 2: Objetivo de cobertura a fallos e impacto de las distintas técnicas a estudiar.

En la Figura 2 se muestran dos alternativas, a las que se orienta este trabajo de tesis, que buscan alcanzar una solución de compromiso entre el impacto sobre el rendimiento del sistema y la cobertura de fallos. La primera de ellas consiste en la utilización de un módulo hardware que se encargue de monitorizar el flujo de instrucciones, y que por tanto sea capaz de detectar errores de control flujo (*Control Flow + Módulo IP*). En el capítulo 3 se proponen varias técnicas novedosas para este propósito, con las cuales se pretende alcanzar un porcentaje de detección de errores de control de flujo cercano al 100%. Por otra parte, mediante la implementación en hardware se pretende que el impacto sobre el rendimiento del sistema sea mucho menor, pudiendo llegar teóricamente a ser nulo.

La segunda alternativa que se ha estudiado en este trabajo consiste en complementar la solución anterior con técnicas de detección de errores de datos implementadas en software. Mediante la implementación de esta segunda alternativa se pretende alcanzar una solución que proporcione una alta cobertura de fallos y un mejor balance entre la cobertura de fallos

y el rendimiento del sistema. Esta solución pretende ser intermedia entre la primera alternativa propuesta (*Control Flow + Módulo IP*) y la técnica utilizada como punto de partida de la tesis, consistente en la duplicación del programa y posterior comprobación vía hardware (*Duplicación + Módulo IP*).

1.2. Objetivos

En este trabajo de tesis doctoral se proponen nuevas técnicas híbridas para la detección de errores en microprocesadores. El objetivo global que se persigue es proporcionar una solución que mejore el compromiso entre la tolerancia a fallos y el impacto sobre el rendimiento del sistema respecto a las soluciones presentes en la literatura. Además, esta solución debe ser no intrusiva, es decir, que no requiera la modificación del microprocesador y debe proporcionar también una baja latencia en la detección de errores.

Este objetivo global se desglosa en los siguientes aspectos:

- Desarrollo de nuevas técnicas de detección de errores de control del flujo en microprocesadores mediante módulos hardware externos. Dichas técnicas serán además implementadas en combinación con técnicas de endurecimiento software existentes en la literatura con el objetivo de alcanzar elevadas tasas de detección de errores.
- Aumento de la observabilidad del sistema mediante la utilización de interfaces de depuración presentes en los microprocesadores modernos. Esto permite realizar una observación del sistema de una manera no intrusiva y con un coste residual, ya que se está llevando a cabo la reutilización de un hardware existente.
- Implementación de las técnicas que se propongan con diferentes microprocesadores y con diferentes aplicaciones software de prueba, para evaluar la capacidad para detectar errores y el impacto sobre las prestaciones y el área del sistema de manera general.
- Análisis de la eficacia de las técnicas propuestas y la dependencia

de las mismas respecto al software utilizado. Para este análisis se realizarán campañas de inyección de fallos lo más exhaustivas que sea posible.

En definitiva, se pretende proponer técnicas híbridas de detección de errores en sistemas basados en microprocesador que alcancen una alta cobertura de detección de errores con un impacto sobre el rendimiento del sistema que sea lo más bajo posible.

1.3. Organización del documento de tesis

El documento de esta tesis doctoral está dividido en seis capítulos. El capítulo dos presenta el estado del arte actual con una revisión de las técnicas existentes para la detección de errores en microprocesadores y una breve introducción a los conceptos relacionados con la tolerancia a fallos. El capítulo tres describe las aportaciones originales desarrolladas en esta tesis doctoral, que consisten en tres técnicas de detección de errores control del flujo en microprocesadores. Los capítulos cuatro y cinco recogen los resultados experimentales obtenidos. Finalmente, en el capítulo seis se describen las conclusiones de la tesis, así como posibles líneas futuras de trabajo.

En la Figura 3 se muestra la organización del documento de tesis. En la columna de la izquierda se muestra el título de cada capítulo, mientras que en la columna de la derecha se describe el concepto principal desarrollado en dicho capítulo.

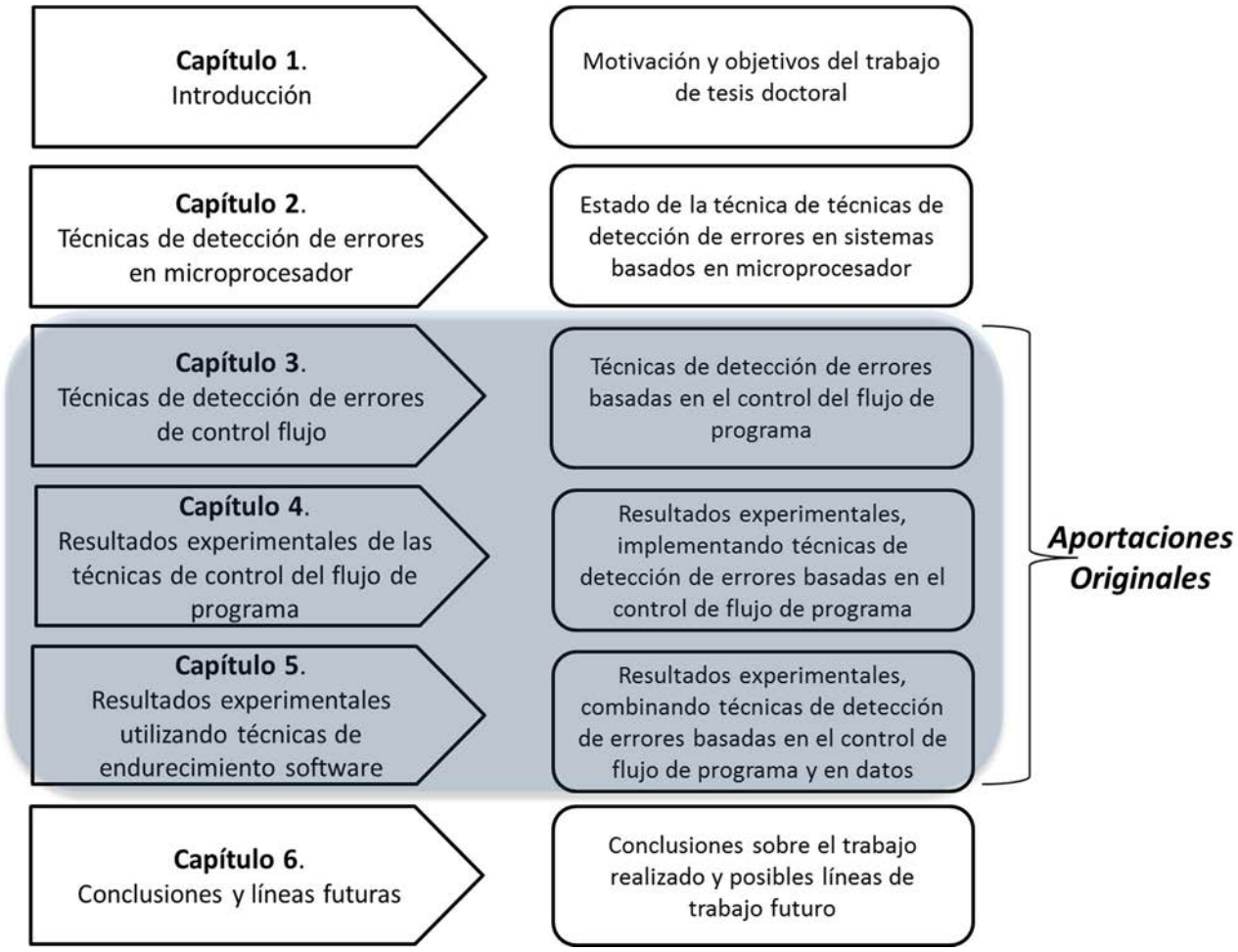

Figura 3: Organización del documento de tesis doctoral.

En el capítulo 2 se revisan las principales técnicas existentes para la detección de errores en microprocesadores. El principal objetivo de este capítulo es mostrar una visión general de la problemática relacionada con la detección de errores en microprocesadores y enmarcar el trabajo presentado en relación con las diferentes soluciones que se proponen en la literatura.

En el capítulo 3 se describen las técnicas de detección de errores control de flujo que se proponen en esta tesis y que constituyen su principal aportación original. Las técnicas propuestas se denominan *Predicción del Contador de Programa*, *Monitorización de Firmas* y *Monitorización Dual*. Dichas técnicas están basadas en la observación del flujo de instrucciones del microprocesador con la finalidad de detectar errores.

En el capítulo 4 se muestran los resultados experimentales obtenidos aplicando las técnicas propuestas en esta tesis doctoral. Los experimentos realizados tienen como objetivo validar dichas técnicas y determinar su capacidad para detectar errores. Se han realizado diversas campañas de inyección de fallos mediante simulación y emulación en *FPGA*, utilizando dos microprocesadores de características distintas, concretamente *LEON3* y *PicoBlaze*.

El capítulo 5 muestra los resultados experimentales obtenidos al aplicar las técnicas de detección de errores de control del flujo propuestas en combinación con técnicas de endurecimiento software. Aunque los errores que afectan al flujo de programa son como regla general los más relevantes, para lograr tasas de cobertura de fallos elevadas es preciso complementarlas con técnicas de detección de errores de datos. Con la finalidad de comprobar la eficacia de las técnicas propuestas y su dependencia respecto al software utilizado, se han realizado nuevos experimentos utilizando software endurecido. Al igual que en el Capítulo 4, los experimentos se han llevado a cabo en dos microprocesadores diferentes, *Picoblaze* y *LEON3*. Sin embargo, en este caso únicamente se han llevado a cabo experimentos de emulación en *FPGA*.

Finalmente, el capítulo 6 muestra las conclusiones extraídas del trabajo realizado así como las posibles líneas de trabajo futuro.

CAPÍTULO 2

TÉCNICAS DE DETECCIÓN DE ERRORES EN MICROPROCESADOR

2. Técnicas de detección de errores en microprocesador

2.1. Introducción

En este capítulo se detallará el estado de la técnica relacionado con el ámbito del conocimiento en el que se enmarca esta tesis doctoral. Esta tesis doctoral se enmarca dentro de la detección de errores en arquitecturas microprocesadores, es por ello que en este capítulo se resumirán tanto la tipología de fallos que estamos tratando de detectar (apartado 2.1.1.) así como las técnicas que se pueden encontrar en la literatura para detección de errores en microprocesadores.

Dentro del campo de detección de errores en microprocesadores tradicionalmente se pueden dividir las técnicas en técnicas hardware, técnicas software y técnicas híbridas [Nico11]. Las técnicas hardware se basan en la modificación del circuito para robustecerlo, estas técnicas están ligadas de forma intrínseca a la arquitectura utilizada y se describen en el apartado 2.3. Las técnicas software se basan en la modificación del software ejecutado por el microprocesador para endurecer la arquitectura en su conjunto. Las técnicas software se describen en el apartado 2.2. Una clara ventaja de las técnicas software frente a las técnicas hardware es que su implementación resulta más sencilla ya que no se modifica el circuito y además por ello no es necesario tener un conocimiento exhaustivo del circuito que por otra parte es desconocido en algunos casos como por ejemplo en un *COTS*. La desventaja clara que presentan las técnicas software frente a otro tipo de técnicas es la imposibilidad de acceder desde el software a todos los elementos arquitecturales y que por lo tanto dificulta un endurecimiento completo de la arquitectura. Por último existe otra posibilidad que son la utilización de forma mixta de las dos técnicas expuestas, hardware y software, tratando de aunar las ventajas que presentan ambas aproximaciones. Un resumen de las técnicas híbridas propuestas en la literatura se detalla en el apartado 2.4.

2.1.1. Tipos de fallos

Los fallos son imperfecciones o defectos en los componentes hardware o software de los circuitos. Estos fallos se manifiestan como errores que pueden causar el malfuncionamiento del circuito [Port07].

Los circuitos tolerantes a fallos son capaces de continuar su normal funcionamiento en presencia de errores hardware o software. Para conseguir que un circuito sea tolerante a fallos es necesario realizar una serie de modificaciones aplicando técnicas de tolerancia a fallos para robustecer el circuito.

Existen numerosas clasificaciones de fallos atendiendo a diversos factores entre los que destacan la etapa del ciclo de diseño y la duración del fallo. Atendiendo a la etapa de diseño los fallos pueden aparecer en todas las etapas que comprenden el diseño y funcionamiento del circuito (diseño, fabricación, operación, etc.). Atendiendo a la duración del fallo, este puede ser permanente, intermitente o transitorio. Los fallos permanentes tienen duración ilimitada mientras que los transitorios e intermitentes tienen duración limitada pero los intermitentes además se repiten en el tiempo. Los fallos transitorios se denominan comúnmente soft errors.

Esta tesis se centrará en fallos no permanentes (soft errors) inducidos bajo los efectos de radiación que ocurren en la etapa de operación del circuito. En aplicaciones terrestres los soft errors son debidos principalmente a dos fuentes de radiación: neutrones y partículas alpha. Los neutrones se generan cuando la radiación cósmica interactúa con la atmosfera terrestre. Las partículas alpha las emiten impurezas radioactivas que están presentes en baja concentración en los chips y encapsulados.

Existen entornos especiales en los cuales la tolerancia a fallos cobra gran relevancia ya que los circuitos están expuestos a radiación y necesitan protegerse para aumentar su fiabilidad. Circuitos que se alejan de la atmosfera terrestre pierden la protección que esta les brinda frente a la radiación. Por lo tanto en el ámbito aeroespacial cobra gran relevancia la tolerancia a fallos y las técnicas utilizadas para conseguir robustecer los

circuitos. En el espacio exterior los circuitos están expuestos a los cinturones de radiación, tormentas solares, viento solar, iones pesados y rayos cósmicos.

Los fallos no permanentes se clasifican de la siguiente forma [Sonz09]:

- *Single Bit Upset* (*SBU*) or *Single Event Upset* (*SEU*): se deben al impacto de una partícula que cambia el estado (*bit-flip*) de un biestable o una celda de memoria.
- *Multiple Bit Upset (MBU):* como el *SBU* pero el cambio se produce en dos o más bits.
- *Single-event transient* (*SET*): el evento (impacto de la partícula) genera un pico de corriente que puede llegar hasta los elementos de memorias y provocar un cambio en ellos.
- *Single-event latchup* (*SEL*): el evento genera un estado de alta corriente. Para reestablecer el circuito es necesario resetearlo. Este tipo de fallo puede producir un fallo permanente.
- *Single-event functional interrupt* (*SEFI*): el evento genera un fallo que afecta a un registro crítico que produce una interrupción en el correcto funcionamiento del circuito. Un *SEFI* siempre provoca un error.

Esta tesis se enmarca dentro del ámbito de la tolerancia a fallos en las arquitecturas microprocesadoras y es por ello que se abordan en este capítulo las diferentes técnicas que se utilizan para ello. Tradicionalmente las técnicas se subdividen teniendo en cuenta dos aspectos: dónde se realiza la modificación para robustecer el circuito (técnicas hardware o software) y que tipo de error se quiere detectar/corregir (errores de datos o de control de flujo).

2.2. Técnicas Software

Las técnicas software se fundamentan en la modificación del código ejecutado por el microprocesador para robustecer la arquitectura en su conjunto. Estas técnicas presentan ciertas ventajas frente a otras

aproximaciones, al tratarse de una modificación software es más sencilla de implementar que cualquier modificación hardware que implica conocimientos específicos de la arquitectura y necesita de un experto para realizarla. Las modificaciones software implican aumento del tamaño de código, que lleva asociado un aumento del tamaño de la memoria que alberga el código y dependiendo del método utilizado, también se puede necesitar espacio en memoria de datos para albergar ciertas estructuras necesarias para el endurecimiento. Este aumento de instrucciones a ejecutar se traduce a su vez en una disminución de la velocidad en la ejecución del código. Ambas desventajas unidas se traducen en una disminución de prestaciones tanto en velocidad como en tamaño de las memorias utilizadas.

Las técnicas software son las candidatas ideales para arquitecturas que no son modificables o son desconocidas, como por ejemplo un COTS.

A continuación se resumirán una selección de técnicas software relevantes que se aplican para robustecer arquitecturas microprocesadores. Las técnicas se han subdivido con la clasificación tradicional utilizada en este campo: técnicas de control de flujo de programa (apartado 2.2.1) y técnicas de datos (2.2.2).

2.2.1. Técnicas de control del flujo de programa

En esta sección se muestran las principales técnicas software utilizadas para endurecer un microprocesador frente a errores que afecten al flujo de programa, en inglés denominado *Control flow error* (*CFE*). Un *CFE* es un error que produce un cambio en el flujo de programa y como consecuencia del mismo el microprocesador ejecuta una instrucción diferente a la instrucción que se ejecutaría si no se hubiera producido error alguno.

Una práctica habitual para aplicar técnicas de control de flujo es dividir los programas en bloques básicos (*BB*). Un bloque básico es un conjunto de instrucciones que son ejecutadas de manera consecutiva sin saltos intermedios. La única instrucción que puede ser de salto es la última instrucción del bloque; y la única instrucción que puede ser el destino de un

salto es la primera. Por tanto, un bloque básico no contiene ninguna instrucción que pueda modificar el flujo de ejecución del programa, excepto como se acaba de comentar la última instrucción. Se denomina cuerpo de un bloque básico al conjunto de las instrucciones del bloque básico salvo la última instrucción siempre que sea de salto. Si la última instrucción no es una instrucción de salto entonces el cuerpo del bloque básico coincide con el bloque básico. Se puede dar el caso de que el cuerpo de un bloque básico esté vacío siempre que el bloque básico esté formado por una única instrucción siendo esta de salto.

Un programa *P* puede ser representado por un gráfico del flujo de control, compuesto por un conjunto de nodos *N* y un conjunto de bordes *B, P={N,B}*, donde $N=\{n_1, n_2, \ldots n_n\}$ y $B=\{b_{i1,j1}, b_{i2,j2}, \ldots b_{im,jm}\}$. Cada nodo n_i representa una sección del programa, la cual puede corresponderse con una única instrucción o con un bloque básico. Cada borde b_{ij} representa el salto desde el nodo n_i al nodo n_j [Yau80]. Considerando un programa cualquiera, es posible definir el conjunto de nodos sucesores de cada uno de los nodos del programa, así como definir el conjunto de nodos predecesores de cada nodo [OhSh02]. Un nodo v_i pertenece al conjunto de sucesores del nodo v_j si y solo si el borde $b_{i,j}$ pertenece al conjunto de bordes. De igual manera un nodo v_j pertenece al conjunto de predecesores del nodo v_i si y solo si $b_{j,i}$ pertenece al conjunto de bordes. Por otro lado, un salto $b_{i,j}$ es ilegal si no está incluido en el conjunto de bordes [Yau80]. Se puede dar el caso que en un programa debido a un fallo se ejecuta un salto $b_{i,k}$ perteneciente al conjunto de bordes en lugar del salto correcto $b_{i,j}$. Este salto a pesar de pertenecer al conjunto de bordes produce una ejecución incorrecta y por tanto es erróneo. Tanto los saltos ilegales y los saltos erróneos pertenecen al grupo de los errores que afectan al flujo de control.

Los errores que afectan al flujo de programa pueden ser divididos en errores que producen saltos entre diferentes bloques y se denominan "*inter-block*" y errores que producen saltos dentro del mismo bloque y se denominan "*intra-block*". Las técnicas de *CFC* para detectar errores *intra-*

block se encargan de controlar que las instrucciones dentro de un mismo bloque se ejecuten en el orden correcto.

Una de las técnicas software más utilizadas para detectar errores que afectan al flujo de programa es la técnica de monitorización de firmas "*Signature monitoring*". En esta técnica la monitorización es realizada a través de las instrucciones regulares del procesador embebidas en el programa bajo ejecución. A cada bloque básico se le asigna una firma durante el tiempo de compilación o a través de un programa especial antes de la ejecución del programa. Durante la ejecución del programa se ejecuta una firma online. Posteriormente se realiza una comparación entre la firma online y la calculada en tiempo de compilación, detectando un error en el flujo de programa en el caso de que difieran. La firma que se calcula en tiempo de ejecución debe ser almacenada en áreas del microprocesador como pueden ser los registros del mismo o en un bloque de memoria de datos.

La tipología de errores que afectan al flujo de programa se puede resumir tal y como se detalla a continuación:

- Tipo 1: Un fallo que causa un salto ilegal desde el final de un bloque básico al inicio de otro bloque básico.
- Tipo 2: Un fallo que causa un salto ilegal pero erróneo desde el final de un bloque básico al inicio de otro bloque básico.
- Tipo 3: Un fallo que causa un salto del final de un bloque básico a cualquier punto del cuerpo de otro bloque básico.
- Tipo 4: Un fallo que causa un salto desde cualquier punto del cuerpo de un bloque básico a cualquier punto del cuerpo de otro bloque básico.
- Tipo 5: Un fallo que causa un salto desde cualquier punto del bloque básico a cualquier otro punto del mismo bloque básico.

Los primeros cuatro tipos de errores representan errores *inter-block* que afectan al flujo de programa, mientras que el último tipo representa errores *intra-block* que afectan al flujo de programa.

Existen numerosos trabajos en la literatura que utilizan técnicas de monitorización de firmas para realizar el control de flujo de los programas ejecutados en arquitecturas microprocesadoras. En los siguientes subapartados se resumen las más destacadas de ellas.

2.2.1.1. ECCA

La técnica para detectar errores en el flujo de programa *ECCA, "Enhanced CCA"* es la versión mejorada de la técnica *CCA "High level control flow checking approach using assertions"* [Mcfe95], [Kana96], [Nair96]. En *CCA* el programa es dividido en bloques básicos a los que se asignan dos identificadores *BID* (*Branch free Interval Identifier*) y *CFID* (*Control Flow Identifier*). El identificador *BID* representa al bloque y es único para cada bloque. El identificador *CFID* representa el flujo de programa permitido y es el mismo para todos los bloques que compartan el mismo bloque previo.

La comprobación del flujo de programa se realiza mediante el cálculo y verificación de los identificadores *BID* y *CFID*. A la entrada de cada bloque el identificador actual *BID* es asignado a una variable *BID*. Una vez que se sale del bloque se comprueba el identificador actual *BID*. Si se ha entrado a un bloque en la mitad del mismo, el valor del identificador actual *BID* no coincidirá y se habrá detectado un error. El identificador *CFID* es utilizado para asegurar que los bloques se han ejecutado en el orden correcto. Los identificadores *CFID* son almacenados en una cola de dos elementos. La cola es inicializada con el identificador del primer bloque. Cada vez que se entra en un nuevo bloque el *CFID* del siguiente bloque es almacenado en la cola. Cada vez que se sale de un bloque, se coge el identificador de la cola y se comprueba que coincida con el identificador *CFID* actual.

A continuación se describen algunos errores en el flujo de control que serían detectados por este método:

- El identificador actual *BID* no coincide con la variable *BID*. Se ha producido un salto ilegal desde el anterior bloque a la mitad del bloque actual.

- Desbordamiento, *Overflow*, en los elementos de la cola. Esta situación tiene lugar cuando se produce un salto ilegal desde la mitad del bloque anterior al inicio del bloque actual.

- *Underflow* en los elementos de la cola. Sucede cuando se produce un salto ilegal desde el final del bloque anterior a la mitad del bloque actual.

- Desajuste *BID*. Se ha producido un salto desde la mitad de un bloque a la mitad del bloque actual.

La técnica *CCA* puede detectar todos los errores simples que afectan al flujo de programa y la mayoría de los errores múltiples. Sin embargo, supone un impacto importante en el incremento del tamaño del programa. En programas con bloques pequeños se incrementa en gran medida el tamaño del programa, debido al gran número de comprobaciones y operaciones a realizar cada vez que se cambia de bloque. Además debido al incremento del tamaño de programa se produce un incremento de la probabilidad de que se produzca un error en el flujo de programa. También hay que tener en cuenta que se produce un incremento importante del área, debido a la necesidad de almacenar en registros los elementos de la cola y los identificadores de cada bloque.

La técnica *ECCA* es la sucesora del método de detección de errores de flujo *CCA* y consigue eliminar muchos de los inconvenientes que presentaba dicho método. A diferencia de la técnica *CCA*, *ECCA* divide el programa en bloques, donde cada bloque está formado por un conjunto de bloques libres de saltos con un único punto de entrada y un único punto de salida. Un punto de salida únicamente podrá saltar a un punto de entrada de otro bloque. A cada bloque se le asigna un número primo único mayor de 2 como identificador del bloque y se le denomina *BID*. Se insertan dos líneas nuevas de código para realizar las comprobaciones del flujo de programa.

La primera línea es una asignación simple ejecutada cuando se entra en el bloque. La segunda línea es también una asignación, ejecutada justo antes de abandonar el bloque. Al Iniciar y finalizar cada bloque se actualiza una variable global con las dos asignaciones antes comentadas y se comparan con los valores calculados en tiempo de ejecución; si hay alguna diferencia se habrá producido un error. Gracias a que un bloque está formado por un conjunto de bloques libres de salto, el número de comprobaciones se ve reducido y por tanto se reduce el tamaño de programa y el incremento del área necesaria. Sin embargo según aumenta el tamaño del bloque, la latencia de detección de errores también aumenta.

En [Alkh99] se propone el siguiente modelo de errores que afectan al flujo de control con la finalidad de medir la eficiencia del método:

- Tipo 1: Fallo que provoca un salto de la mitad de un bloque a otro.
- Tipo2: Fallo que provoca un salto a la mitad de un bloque.
- Tipo3: Fallo que provoca el salto a un bloque ilegal.

Los errores de datos, como por ejemplo errores en las condiciones de los saltos, no están cubiertos por la técnica *ECCA*. El método *ECCA* es capaz de detectar todos los errores de flujo simples y la mayoría de los errores múltiples, sin embargo, no sería capaz de detectar un error múltiple cuando este provoca un salto de la mitad de un bloque a la mitad de otro bloque de manera sucesiva; tampoco sería capaz de detectar un error múltiple cuando el bloque de inicio sea el mismo que el bloque de fin. Este último caso puede ser considerado como un error de flujo dentro del mismo bloque y por tanto no puede ser detectado.

Los resultados experimentales realizados en [Alkh99] muestran que esta técnica requiere un mínimo incremento en el área y en el tamaño de programa para conseguir una cobertura de detección de errores alrededor del 98%. Realizando una selectiva selección de los bloques de programa se pueden alcanzar unas reducciones en el impacto del rendimiento y el área respecto a la técnica *CCA* que hacen que la técnica *ECCA* sea ideal para

sistemas en tiempo real. Además la arquitectura portable de la técnica *ECCA* permite su utilización en todo tipo de sistemas distribuidos.

2.2.1.2. CFCSS

CFCSS [NOh02a] es un método software para la detección de errores de flujo de programa que utiliza firmas para la detección de dichos errores.

El método de monitorización de firmas es un método para la detección de errores de flujo en el que una firma, asociada a un bloque de instrucciones, es calculada en tiempo de compilación y almacenada para posteriormente poder ser comparada con otra firma calculada en tiempo de ejecución. La asignación de las firmas puede realizarse de manera arbitraria o derivada del código binario o las direcciones de las instrucciones. *SIC* "*Structural Integrity Checking*" [DJLu82] utiliza un método de monitorización de firmas con asignación arbitraria, mientras que *PSA* "*Path Signature Analysis*" [Namj82], *SIS* "*Signature Instruction Streams*" [Shen82], *ASIS* "*Asynchronous SIS*" [Eifer84], *CMS* "*Continuous Signature Monitoring*" [Wilk89], [Wilk90] y *OSLC* "*On-line Signature Learning and Checking*" [Made92] utilizan una asignación de firmas derivadas del código binario o las direcciones de las instrucciones.

Muchas técnicas de monitorización de firmas utilizan un hardware dedicado para el cálculo on-line de las firmas y compararlas con las firmas calculadas en tiempo de ejecución. Para llevar a cabo el cálculo de las firmas y las comparaciones se utiliza un módulo hardware denominado Watchdog processor [DJLu80], [Mahm85].

CFCSS comprueba el flujo de programa utilizando un registro dedicado que almacena la firma en tiempo real *G* del nodo actual (nodo que contiene la instrucción que se está ejecutando). En tiempo de compilación se asigna a cada bloque básico un identificador y una firma única *S*. En una ejecución normal libre de errores, la firma calculada en tiempo real debería de ser igual a la firma del bloque que se esté ejecutando, es decir, *G* debería de ser igual a *S*. Si *G* contiene un valor diferente al de la firma asociada al nodo actual se habrá producido un error.

Cuando se pasa de un bloque a otro, se calcula la firma en tiempo real G utilizando una función de firma *f*. La función *f* actualiza G utilizando dos valores, la firma del nodo previo (nodo desde el que se produce el salto) y la firma del nodo actual (nodo destino del salto). La operación elegida para implementar la función *f* es la *XOR*, ya que presenta mejor comportamiento ante operaciones de comprobación y generación de firmas que otras operaciones de la *ALU*. Operaciones como *AND*, *OR* y *XOR* usan menos puertas que otras operaciones de la ALU como la suma y la multiplicación, por lo tanto tienen menos posibilidades de tener un error durante la operación en la *ALU*, y la probabilidad de que la firma esté bien calculada es mayor. En las operaciones *AND* y *OR*, dadas una entra y una salida determinada no se puede determinar de manera unívoca la otra entrada. La operación *XOR* no presenta este problema, y por esta razón es utilizada para el cálculo de las firmas.

Todos los bloques básicos (nodos) son identificados y numerados. Cada bloque básico tiene asignado una única firma que es calculada en tiempo de compilación y almacenada para su posterior comprobación. Durante la ejecución, cada vez que el control entre en un nuevo nodo, la firma en tiempo real G es actualizada utilizando la función *f* y el valor previo de G y la firma del nuevo nodo como argumentos de la función. Si la nueva firma G calculada es igual que la firma del nuevo nodo, no se ha producido ningún error en el flujo de programa y por tanto se ejecutan las instrucciones del nuevo nodo, en caso contrario, se ha producido un error de flujo y el control es transferido a una rutina para el tratamiento del error.

Con la finalidad de realizar el control del flujo de programa se añaden una serie de instrucciones de comprobación al inicio de cada bloque básico. Por tanto, el bloque estará formado por una serie de instrucciones a las que se han añadido instrucciones de comprobación al inicio de cada uno de los bloques. Las instrucciones de comprobación se pueden dividir en dos partes:

- La función que genera la firma en tiempo de ejecución.

- La instrucción de salto que compara la firma calculada en tiempo de ejecución con la firma del bloque básico.

A priori con el cálculo de la firma y su posterior comprobación es suficiente para detectar un salto ilegal. Sin embargo, hay casos en los que se asigna una misma firma a distintos nodos, esto sucede cuando dos nodos comparten un mismo nodo destino. En este caso, en el que hay varios nodos predecesores con la misma firma, se utiliza una nueva función D. Después de generar G se hace una XOR de G con D para obtener el valor de la firma del nodo de destino. D debe de tener un valor que haga esta operación igual al valor de la firma del nodo de destino. Por tanto, si un nodo puede saltar a un nodo con varios predecesores debe de tener una instrucción adicional, para calcular D, que se añadirá a las dos instrucciones de comprobación anteriormente descritas. Si se produce el salto al nodo con varios predecesores se realiza la XOR de la G con D; en caso contrario D es ignorada.

En el caso de que múltiples nodos compartan nodos con varios predecesores se puede producir un efecto de "aliasing" entre saltos legales e ilegales de manera que hagan que el error de flujo sea indetectable.

Una serie de experimentos realizados en [NOh02a] demuestran que la técnica *CFCSS* incrementa la capacidad de detección de errores en un factor de 10. Teniendo en cuenta que cada nodo tiene entre dos y cuatro instrucciones adicionales y que el tamaño medio de un bloque básico está entre siete y ocho [Henn96] el incremento del área está comprendido entre el 25% y el 43%. En principio cada nodo tiene una instrucción que compara la firma calculada en tiempo de ejecución con la firma de cada nodo. Sin embargo, puede ser que no sea crítico detectar el error de inmediato, y en este caso podría retrasarse la comprobación. Una vez que se produce un salto ilegal, la firma calculada en tiempo de ejecución no se corresponde con la firma del nodo, ni se corresponderá con la firma de los siguientes nodos. De esta manera, las instrucciones de comprobación pueden ubicarse únicamente en los nodos en los que sea importante detectar los errores en el flujo de programa de manera inmediata. Si se ponen

instrucciones de comprobación en todos los nodos el incremento del tamaño de programa es de un 30% y un 37,5 % en los dos ejemplos que se proponen en [Alkh99] y una latencia de 4,5 instrucciones de media en ambos casos. Sin embargo, si se ponen las instrucciones de comprobación únicamente en los nodos más críticos, el incremento de área puede reducirse hasta un 19% y 27% respectivamente con una latencia de 22,8 instrucciones y 21,8 instrucciones en cada uno de los casos.

2.2.1.3. CEDA

La técnica *CEDA* [Vemu06] es una técnica software de control del flujo de programa que restringe las firmas que pueden ser asignadas a un bloque básico siguiendo una serie de reglas.

A continuación se muestran una serie de definiciones que se utilizan a lo largo de la explicación del método:

- Tipos de nodos: Un nodo (*Ni*) es del tipo A si tiene múltiples predecesores y al menos uno de sus predecesores tiene múltiples sucesores. Un nodo es de tipo X si no es de tipo A.
- Registro de la firma (*S*): firma calculada en tiempo de ejecución que permite la monitorización del programa.
- Firma Nodo (*NS*): valor esperado de la firma en cualquier punto dentro del nodo en una ejecución correcta del programa.
- Firma Nodo de Salida (*NES*): valor esperado de la firma en un nodo de salida en una ejecución correcta del programa.

Se añaden instrucciones al programa de manera que se calcula continuamente una firma on-line con el fin de monitorizar en todo momento la ejecución del programa. Para alcanzar este propósito se analiza el programa a ejecutar y se representa como un gráfico de programa. Se asigna a cada nodo una firma nodo *NS* y una firma nodo de salida *NES* de acuerdo con unas reglas que se detallan más adelante. Existen dos parámetros, d_1 y d_2, asociados con cada uno de los nodos, que son utilizados

para el cálculo on-line de la firma *S*. Estos parámetros tienen unos valores, asignados en tiempo de compilación, que hacen que la firma calculada en tiempo de ejecución coincida con los valores esperados siempre que no se produzca un error. Lo que se busca es encontrar unos valores de *NS* y *NES* de cada nodo de manera que cualquier error en el flujo de programa se traduzca en una diferencia en la firma on-line respecto al valor esperado de la misma. En este método únicamente se tienen en cuenta aquellos errores en el flujo de programa que producen un salto ilegal entre nodos.

Durante la ejecución del programa la firma es actualizada dos veces en cada nodo. Al inicio de la ejecución del nodo la firma se actualiza siguiendo las siguientes fórmulas:

- $S = S\ AND\ d_1(N_i)$ si el nodo es de tipo *A*.
- $S = S\ XOR\ d_1(N_i)$ si el nodo es de tipo *X*.

Como resultado de estas operaciones el valor de la firma debería ser igual al esperado. Al final de la ejecución de cada nodo la firma se actualiza siguiendo la siguiente fórmula:

- $S = S\ XOR\ d_2(N_i)$

En ciertos puntos del programa, a los que se les denomina "*check points*", se añaden instrucciones de comprobación. Una instrucción de comprobación detecta un error en el flujo de programa que se haya producido anteriormente a dicha instrucción. En una instrucción de comprobación, la firma actual *S* se compara con el valor esperado, y se llama a una rutina de atención al error en caso de que se produzca un error de flujo.

En la Figura 4 se muestra la necesidad de hacer distinción entre nodos de tipo *A* y nodos de tipo *X*. Supongamos que el nodo N_8 (nodo tipo A) es tratado como un nodo de tipo *X* y se realiza, por tanto, una operación tipo *XOR* al inicio del nodo. En este caso la firma esperada en N_8 es independiente de si N_8 se ha ejecutado después de N_6 o después de N_7. Esto significa que la firma de salida esperada en N_6 y N_7 es la misma. Por tanto, el valor de la firma *S* en N_9 es igual independientemente de que antes se haya

ejecutado N_6 o N_7. En este caso no se puede distinguir entre un salto de N_7 a N_9 y un salto ilegal de N_6 a N_9.Esta situación es la que obliga a hacer una distinción entre los nodos de tipo A y los nodos de tipo *X*, y utilizar la operación *AND* para un nodo de tipo *A* al inicio del nodo. De esta manera las firmas de salida de los nodos N_6 y N_7 son distintas y por tanto se podrá detectar un salto ilegal de N_6 a N_9.

Un requisito importante para evitar el efecto de *aliasing* es que las firmas esperadas al final del nodo y en el nodo sean únicas.

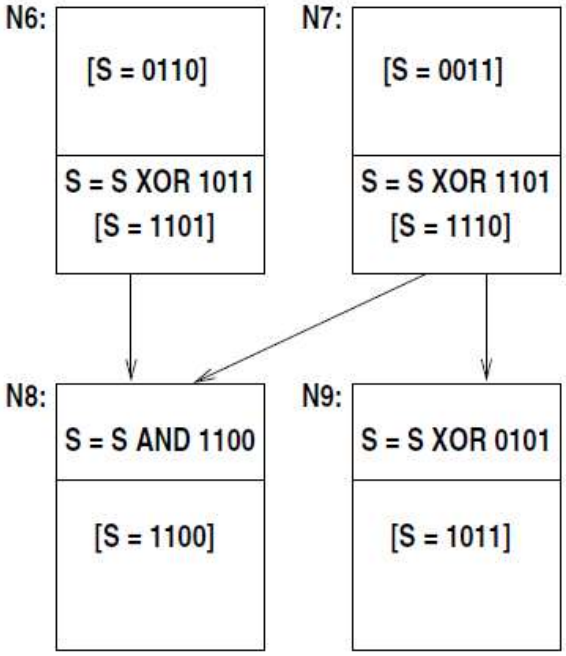

Figura 4: Distinción entre nodo tipo A y nodo tipo X.

Este requisito conlleva que un salto ilegal produzca una diferencia entre la firma esperada y la calculada on-line nada más producirse el salto. Otro requisito para detectar errores es que los valores de las firmas después de un salto se mantengan hasta el siguiente punto de comprobación.

A continuación se muestran una serie de definiciones utilizadas para realizar la explicación del algoritmo de asignación de las firmas:

- Red *(Net):* una red es un conjunto de nodos en la que todos los nodos comparten algún predecesor. Cada nodo únicamente puede pertenecer a una red.

- Red de predecesores *(net_pred):* es el conjunto de predecesores de cada uno de los elementos de una red.
- Conjunto de firmas relacionadas *(A_sig):* es un conjunto formado por las firmas de fin de nodo *(NES)* de los nodos pertenecientes a la red de predecesores y las firmas de nodo *(NS)* de todos los nodos de tipo A pertenecientes a la red.

En *CEDA* la parte alta y baja de la firma esperada se asigna de manera separada. La parte alta de las *NS* y las *NES* se asignan como se muestra a continuación. Para cada red la parte alta de todas las firmas pertenecientes al conjunto *A_sig* tienen asignado el mismo valor. Para el resto de las firmas se asigna una parte alta única y diferente. De esta manera la parte alta de la firma *S* sólo tendrá el mismo valor si el salto ilegal se produce a uno de los nodos de la misma red que tiene una firma perteneciente al conjunto *A_sig*.

Para asignar la parte baja de las firmas, se definen una serie de reglas a continuación. En primer lugar se define *Zl*(*Num*) como el conjunto de bits cero en la parte baja del número binario *Num*.

- *Zl(S1) ≠ Zl(S2)* donde *S1* y *S2* pertenecen al conjunto *A_sig*. La única excepción para esta regla es cuando *S1* y *S2* son firmas de salida de unos nodos que comparte el mismo nodo predecesor de tipo X, en este caso *Zl(S1) =Zl(S2).* Con esta regla se asegura que la parte baja, de las firmas que tienen la misma parte alta, será diferente.
- *Zl(NES(N2)) C Zl(NS(N1))* donde *N1* es un nodo de tipo *A* y *N2* pertenece a la red de predecesores de N1.
- *Zl(NES(N2)) ⊄ Zl(NS(N1))* donde *N1* es un nodo de tipo *A* y *N2* pertenece al conjunto *A_sig* y no pertenece a la red de predecesores de *N1*.

Las partes bajas de todas las firmas que todavía no están asignadas (*NS* de los nodos tipo *X*) son asignados con valores únicos. Al final se asignan los valores de d_1 y d_2 de cada nodo. Si N_1 es de tipo *A*, se asignarán valores '1' a toda la parte alta de d_1 y la parte baja será igual a la parte baja de $NS(N_1)$.

Si N_1 es de tipo *X*, d_1 es igual a la *XOR* de $NS(N_1)$ y $NES(N_2)$ donde N_2 es uno de los predecesores de N_1. El valor de d_2 es igual a la *XOR* de $NS(N_1)$ y $NES(N_1)$.

En [Vemu06] se han llevado a cabo una serie de experimentos que demuestran la eficacia de este método. En la Tabla 2 se muestran los resultados obtenidos para distintos benchmark.

Benchmark	Errores no Detectados (%)	Impacto sobre Rendimiento (%)
Parser	1,1	13,79
Gzip	0,6	57,8
Ammp	0,2	3,15
Twolf	0,6	9,8
Equake	0,5	17,9

Tabla 1 : Resultados experimentales CEDA.

Los experimentos realizados en [Vemu06] demuestran que *CEDA* presenta una elevada cobertura de detección de errores con un impacto sobre el rendimiento pequeño.

2.2.1.4. YACCA

La técnica *YACCA* (*Yet Another Control Flow Checking Approach*) [Golo03] asigna a cada bloque básico dos identificadores únicos *I1* y *I2*. El identificador *I1* es asociado con la entrada del bloque básico correspondiente, mientras que el identificador *I2* es asignado a la salida del bloque básico.

Durante la ejecución del programa se utiliza una variable, a la que se le denomina *code*, que se encarga de almacenar la firma online. La variable *code* es actualizada por medio de una aserción *set* al valor del identificador de entrada *I1* al comienzo de cada bloque básico; y al valor del identificador de salida *I2* al final de cada bloque básico. Antes de actualizar

la firma se realizan una serie de comprobaciones sobre la misma para determinar si contiene el valor correcto. Al comienzo de cada bloque básico se comprueba que el valor de la firma online almacenada en la variable *code* se corresponda con el identificador *I2* de alguno de los bloques básicos que forman parte del conjunto de predecesores. Mientras que al final de cada bloque básico se comprueba que el valor de la variable *code* se corresponda con el identificador *I1* del bloque básico destino. La variable *code* es actualizada siguiendo la siguiente fórmula: *code* = (*code* & *M1*) $\oplus$ *M2*. *M1* es una constante cuyo valor depende del conjunto de los valores predecesores posibles de la variable *code*. *M2* es una constante cuyo valor depende del identificador que debería ser asignado a la variable *code,* y de los valores predecesores posibles de la variable *code*.

La representación binaria de *M1* contiene el valor 1 en los bits que tengan el mismo valor en todos los identificadores $I2_j$ del conjunto de bloques básicos predecesores, y el valor 0 en los bits que tengan diferentes valores en esos identificadores. La operación (*code* & *M1*) permite actualizar el valor de la variable *code* al mismo valor *I* de cualquiera de los valores predecesores posibles de la variable *code*. Por lo tanto, realizando la *XOR* de *M2* e *I* se consigue obtener el valor del identificador *I1*. Los bloques básicos deben ser elegidos de tal manera que el nuevo valor de la variable *code* sea igual al valor objetivo si y solo si el valor antiguo de la misma sigue el gráfico del flujo de programa, por ejemplo, la operación ($I2_j$ & *M1*) no debería devolver el valor *I* si el bloque básico no pertenece al conjunto de predecesores.

El test de comprobación que se realiza al comienzo de un bloque básico se implementa de la siguiente manera:

ERR_CODE |= ((*code* != $I2_{j1}$)&& (*code* != $I2_{j2}$) && (...) && (*code* != $I2_{jn}$)

La variable *ERR_CODE* es una variable especial del programa que es activada cuando se produce un error en el flujo de programa. Esta variable es inicializada a cero cuando se inicia la ejecución del programa. Al final de cada bloque básico se realiza la siguiente comprobación:

ERR_CODE |= (*code* != $I1_{j1}$)

Con la finalidad de detectar saltos incorrectos esta comprobación es repetida para cada salto condicional.

Los autores han realizado una serie de análisis experimentales que demuestran que la técnica *YACCA* alcanza una buena cobertura a fallos. Al implementar el método al sistema bajo análisis un número muy limitado de fallos produjeron error; además gracias a los experimentos realizados se demuestra que la técnica *YACCA* es una de las principales alternativas para detectar errores en el flujo de programa. La técnica cubre todos los errores simples de errores tipo 1, tipo 2, tipo 3 y tipo 4.

2.2.2. Técnicas de datos

Si consideramos los posibles efectos que puede producir un fallo en un sistema basado en microprocesador, un tipo de fallo común es el que afecta a los datos de un programa. En este caso el fallo fuerza al programa a proporcionar unos resultados erróneos, mientras que se sigue el mismo flujo de ejecución que en un caso en el que no se ha producido fallo alguno. En esta sección se describen las principales técnicas software destinadas a detectar errores en los datos del programa. La adopción de estas técnicas es generalmente costosa en términos de tamaño de memoria necesaria para la implementación y aumento del tiempo de ejecución del programa, sin embargo, generalmente ofrecen una elevada cobertura a fallos que afectan a los datos del programa. Estas técnicas se pueden dividir principalmente en dos grupos: técnicas basadas en la duplicación de la ejecución a distintos niveles y técnicas basadas en la introducción de una serie de sentencias que comprueban la consistencia de los datos, a las que se les denomina aserciones.

2.2.2.1. Duplicación computacional

Estas técnicas se basan principalmente en la idea de que las operaciones realizadas sobre el código pueden ser duplicadas y los resultados que se producen comparados, permitiendo de esta manera la

detección de fallos. La duplicación puede llevarse a cabo a diferentes niveles: duplicación a nivel de instrucción, duplicación de un conjunto de instrucciones, duplicación a nivel de procedimientos o incluso duplicación a nivel de programa.

Si tenemos en cuenta el menor nivel de duplicación tenemos la duplicación a nivel de instrucción, donde se duplica una instrucción de manera individual. En este caso, la instrucción duplicada podrá ejecutarse después de la instrucción original, llevando a cabo una operación igual a la original, o realizando una operación modificada respecto a la versión original. Por otro lado, el mayor nivel de duplicación consiste en duplicar el programa completo. El programa duplicado puede ser ejecutado después de que el programa original complete su ejecución o de manera concurrente. Independientemente del nivel de duplicación estas técnicas son capaces de detectar fallos gracias a una rutina de comprobación que se encarga de comparar ambas versiones de la ejecución.

En primer lugar se van a describir una serie de técnicas que realizan una duplicación a nivel de instrucción. Posteriormente se describirán diferentes técnicas aumentando el nivel de duplicación. Una técnica sencilla que utiliza una duplicación a nivel de instrucción y que alcanza una importante capacidad en la detección de errores es la descrita en [Reba99]. Esta técnica está basada en la utilización de redundancia en los datos y en el código. Se realizan una serie de transformaciones en el código a alto nivel para implementar la redundancia. El código modificado es capaz de detectar errores que afectan tanto a los datos como al flujo de programa. Para conseguir este objetivo se duplica cada una de las variables del programa y se realizan comprobaciones de consistencia cada vez que se produce una operación de lectura. Por otro lado hay una serie de transformaciones que se centran en errores que afecten al código. Por un lado se duplica el código que implementa cada una de las operaciones de escritura y por otro lado se realizan nuevamente comprobaciones de consistencia de las operaciones ejecutadas.

Una de las principales ventajas de este método es que es totalmente independiente del hardware y por tanto, puede ser implementado directamente sobre el código a alto nivel [Reba01] y posiblemente pueda complementar otras técnicas de detección de errores. Las reglas únicamente están relacionadas con las variables definidas y usadas por el programa, y son aplicadas al código a alto nivel sin tener en cuenta si las variables están almacenadas en la memoria, en la caché o en un registro del microprocesador. Es importante destacar que la capacidad de detección de las reglas aplicadas es considerablemente elevada, detectando errores que afectan a los datos sin ningún tipo de limitación respecto al número de bit afectados o la localización de los mismos. Las reglas básicas que hay que aplicar en esta técnica se pueden resumir en las siguientes:

- Regla 1: cada variable x debe ser duplicada, siendo x0 y x1 los nombres de ambas copias.
- Regla 2: cada operación de escritura realizada sobre la variable x debe ser igualmente llevada a cabo sobre las copias x0 y x1.
- Regla 3: después de cada operación de lectura de la variable x, debe realizarse una comprobación de consistencia sobre las réplicas x0 y x1, y se debe activar la rutina de detección de errores si hay alguna diferencia entre ambas variables.

Las comprobaciones de consistencia tienen que ser realizadas inmediatamente después de las operaciones de lectura con la finalidad de evitar que el error se propague. Igualmente deben realizarse comprobaciones de consistencia cuando una variable aparezca en cualquier expresión utilizada como condición de un bucle o un salto, permitiendo por tanto, la detección de errores que afectan al flujo de programa. Un fallo que afecta a una variable durante la ejecución de un programa es detectado en cuanto dicha variable es utilizada como operando de una instrucción. La latencia de detección es igual a la distancia temporal entre el momento en el que produce el fallo y la primera

operación de lectura posterior al fallo. Los fallos que afectan a variables después de su último uso en el programa no son detectados, pero tampoco producen ningún error en la ejecución. En la Tabla 2 se muestran dos ejemplos de las modificaciones realizadas al código al implementar las reglas de transformación.

Código original	Código modificado
a = b	$a_0 = b_0$ $a_1 = b_1$ si $b_0 \neq b_1 \rightarrow$ error
a = b + c	$a_0 = b_0 + c_0$ $a_1 = b_1 + c_1$ si $b_0 \neq b_1$ o $c_0 \neq c_1 \rightarrow$ error

Tabla 2: Ejemplo de modificación del código al aplicar las reglas de transformación.

En [Chey00] se muestran una serie de experimentos en los que se ha llevado a cabo un endurecimiento software implementando cada una de las reglas de transformación. Los resultados muestran que implementando dicha técnica se alcanza una solución bastante completa respecto a la capacidad de detectar errores, sin embargo, se produce un incremento elevado del área del sistema y del tiempo de ejecución.

En [Bens00] se propone una técnica similar, en la que se seleccionan un subconjunto de variables para ser duplicadas, en lugar de duplicar la totalidad de las variables. De esta manera se consigue disminuir en gran medida el coste que supone implementar la técnica a expensas de una pequeña reducción en la capacidad para detectar errores. Otro método basado en la duplicación de variables en el que se trata de disminuir el impacto sobre el rendimiento del sistema es el que se presenta en [Nico03]. Se propone en primer lugar clasificar las variables en dos categorías teniendo en cuenta las relaciones que existen entre las mismas:

- Variables intermedias: se utilizan para el cálculo de otras variables.
- Variables finales: no forman parte del cálculo de otra variable.

Una vez determinadas las relaciones entre variables, se duplican todas las variables que forman parte del programa. Cada operación realizada en una variable es realizada también en su réplica. Por tanto, las relaciones que siguen las variables replicadas son las mismas que siguen las variables originales. Después de cada operación de escritura sobre una variable final se realiza una comprobación de consistencia entre la variable original y su réplica. La técnica consiste en una serie de reglas que se aplican al código a alto nivel:

- Regla 1: Identificación de la relación que existe entre las diferentes variables del programa.
- Regla 2: Clasificación de las variables de acuerdo con su rol dentro del programa: variables intermedias y variables finales.
- Regla 3: Todas las variables del programa deben ser duplicadas.
- Regla 4: Todas las operaciones que se realicen a lo largo de la ejecución del programa deben ser realizadas en ambas copias de las variables.
- Regla 5: Siempre que se realice una escritura en una variable las dos copias deben ser comparadas con el fin de buscar posibles diferencias en las dos copias. En caso que las copias de la variable final sean diferentes se habrá detectado un error en los datos.

Implementando este conjunto de reglas en combinación con reglas para detectar errores que afectan al flujo de programa se consigue alcanzar una cobertura a fallos importante, reduciendo el impacto sobre el rendimiento del sistema respecto a la solución propuesta en [Reba99].

Otro método basado en un concepto similar y llamado *EDDI* (*Error-Detection by Duplicated Instructions*) es el que se presenta en [NOh02c]. *EDDI* pretende explotar la arquitectura de microprocesadores superescalares modernos para conseguir mejorar la tolerancia a fallos. Los autores de *EDDI* proponen modificar las instrucciones originales en el código fuente en ensamblador, duplicando registros y variables también. Al

ejecutar el código endurecido en una arquitectura superescalar el coste asociado a la duplicación es menor del doble respecto al código original, ya que se aprovechan unidades funcionales dentro del pipeline que estaban ociosas.

Si aumentamos el nivel de duplicación nos encontramos con la técnica *SPCD* [NOh02b] (*Selective Procedure Call Duplication*) la cual está basada en duplicar la ejecución de procedimientos. Cada procedimiento es llamado dos veces con los mismos parámetros; los resultados producidos por cada una de las dos ejecuciones son almacenados y comparados con la finalidad de detectar errores en alguna de las ejecuciones. El objetivo de esta técnica es mejorar la seguridad del sistema detectando errores transitorios en el hardware, y teniendo en cuenta el consumo de energía y el impacto sobre el rendimiento del sistema. *SPCD* minimiza la energía de disipación reduciendo el número de ciclos de reloj, accesos a la memoria caché y accesos a memoria seleccionando los procedimientos a duplicar, en lugar de duplicar cada instrucción. El número de ciclos de reloj es menor ya que el número de comprobaciones se ve reducido a comprobar la ejecución una vez que se han ejecutado el procedimiento original y su réplica. El tamaño del código también se ve reducido ya que el código del procedimiento no es duplicado. El consumo de energía puede ser reducido realizando la búsqueda de instrucciones desde la caché o moviendo instrucciones de la memoria a la caché. Además, reduciendo el número de comparaciones se reduce el número de accesos a la caché de datos y a la memoria, reduciendo el consumo de energía.

Hay que buscar una solución de compromiso entre la latencia y la reducción del consumo de energía. Para reducir el consumo de energía hay que reducir el número de comprobaciones; y cuanto menor sea el número de comprobaciones mayor será la latencia. La menor latencia de detección es alcanzada con la duplicación a nivel de instrucción. En la duplicación a nivel de procedimientos la comparación se post-pone hasta que el procedimiento es ejecutado dos veces. Por tanto, el peor caso de latencia se corresponde con el tiempo de ejecución del procedimiento original y el

procedimiento duplicado más el tiempo que se tarda en ejecutar las comparaciones.

Con el objeto de determinar la capacidad para detectar errores de la técnica *SPCD* los autores han llevado a cabo una serie de experimentos de inyección en los que se han inyectado fallos en el sumador. Los resultados que proporcionan los autores muestran una integridad de los datos del 100%, es decir, la salida siempre es correcta. Además se observa como al aumentar la latencia de detección se reduce el consumo de energía. Por otro lado, el número de fallos detectados disminuye según aumenta la latencia de detección, sin embargo, los fallos que no son detectados no producen ningún error ya que no afectan a los resultados finales. Con la finalidad de evaluar la efectividad del método en términos de consumo de energía, *SPCD* es comparado con el programa endurecido llevando a cabo una duplicación a nivel de instrucción al aplicar la técnica propuesta en [NOh02c]. Los resultados obtenidos muestran que la técnica *SPCD* disminuye el consumo de energía en un 25% respecto al consumo de energía requerido en una técnica que lleva a cabo una duplicación a nivel de instrucción.

Como se ha comentado anteriormente el mayor nivel de duplicación consiste en duplicar la ejecución del programa completo. Esta duplicación puede ser simultánea utilizando varios microprocesadores o adoptar una redundancia temporal en el mismo microprocesador ejecutando múltiples veces el programa. La redundancia temporal es utilizada para detectar errores en el hardware subyacente. Una de las técnicas basadas en este concepto de redundancia temporal es la denominada *Virtual Duplex System* (*VDS*). *VDS* alcanza la duplicación temporal ejecutando dos programas que realizan las mismas tareas y utilizando las mismas entradas. Una de las principales ventajas de esta técnica es que puede ser implementada utilizando un único procesador, el cual es el encargado de ejecutar dos veces el mismo programa. Por contra, una de las principales desventajas que presenta es la degradación del rendimiento que se produce como consecuencia de la repetición de tareas. Existen diversas posibilidades en cuanto a la duplicación. Una de las opciones consiste en

ejecutar el programa entero dos veces; el programa duplicado empezaría al finalizar el original. La segunda opción es ejecutar ambos programas en pequeñas rondas saltando de un programa a otro al finalizar cada una de las rondas. Esta segunda opción supone una mayor degradación en el rendimiento, sin embargo, puede ser usada para comparar resultados intermedios con la finalidad de reducir la latencia.

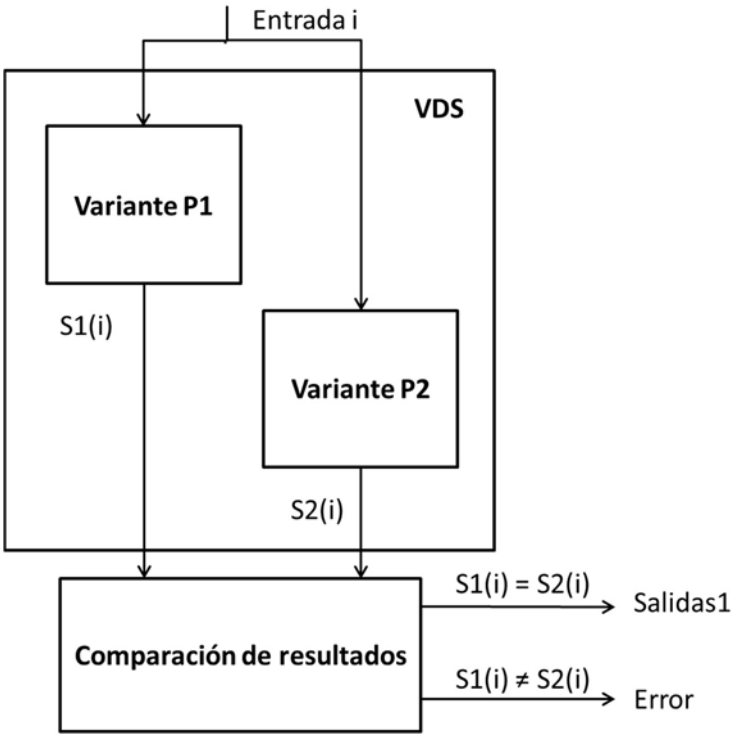

Figura 5: Estructura de la técnica VDS.

En la Figura *5* se muestra la estructura del funcionamiento de la técnica *VDS*. A cada versión del programa se le denomina variante. El sistema mostrado en la Figura 5 consta de dos variantes P1 y P2 que calculan las salidas S1 y *S2* respectivamente, para una entrada *i* dada. En ausencia de fallos las dos salidas *S1* y *S2* serán iguales, sin embargo, si un fallo afecta a una de las dos variantes podrá ser detectado comparando las salidas de las mismas.

El tipo de fallo que puede detectar un *VDS* depende de la diversidad de técnicas usadas para generar el *VDS.* Por ejemplo, si dos equipos diferentes desarrollan variantes de un mismo programa el *VDS* resultante podría tener la habilidad de detectar fallos de implementación y fallos de especificación.

Sin embargo, si se utilizan dos compiladores diferentes para generar las variantes partiendo de un mismo código fuente, el *VDS* resultante tendrá la capacidad de detectar fallos derivados de la compilación. La capacidad para detectar errores de variantes de programas, generadas de manera diversificada, ha sido investigada en [Echt90]. La idea principal es que dos programas que han sido generados de diferente manera utilizan diferentes partes del hardware del microprocesador de diferentes maneras y con datos diferentes para realizar la misma tarea. Las variantes pueden ser generadas manualmente mediante la aplicación de diferentes técnicas de diversidad. Sin embargo, se han propuesto diferentes métodos algorítmicos con la finalidad de obtener variantes de una efectiva. En [Enge97] se presenta un método que se basa en la transformación de cada instrucción de un programa en instrucciones modificadas, manteniendo el algoritmo fijo. Las transformaciones están basadas en diferentes representaciones de los datos. Al haber diversas representaciones de los datos se requiere modificar las instrucciones que van a ser ejecutadas; se debe generar una nueva secuencia de instrucciones que calculen el resultado del programa original con la representación modificada.

En [Joch02] se propone un método en el que las variantes son generadas de manera automática. La herramienta propuesta es capaz de generar dos códigos en ensamblador diferentes pero semánticamente equivalentes. Para ello utiliza un conjunto de normas que aplica a las transformaciones. Algunos ejemplos de estas reglas se muestran a continuación:

- Cambiar instrucciones de salto por otras combinaciones de salto equivalentes.
- Sustitución de una operación de multiplicación por una subrutina que realice la multiplicación de una manera distinta.

Una de las técnicas más innovadoras utilizadas para mejorar el rendimiento de microprocesadores superescalares es la técnica *Simultaneous multithreading* (*SMT*). Un sistema *SMT* permite múltiples hilos de

ejecución independientes. La técnica *VDS* puede ser implementada de una manera eficiente en un sistema *SMT*, ejecutando dos hilos en paralelos, cambiando la redundancia temporal por redundancia espacial [Rein00]. La técnica *Active-Stream/Redundant-Stream Simultaneous multithreading* (*AR-SMT*) propuesta en [Rote99] ejecuta dos copias concurrentes del programa totalmente independientes utilizando los mismos recursos del procesador. El *pipeline* entero del microprocesador es conceptualmente duplicado. Al ejecutar un programa simple en un microprocesador superescalar es común que haya partes del programa que no utilicen todos los recursos del microprocesador, por tanto, es posible compartir los recursos entre varios programas alcanzando una mayor utilización de los recursos del microprocesador. Mejorar la utilización de recursos reduce el tiempo requerido para ejecutar todos los hilos, sin embargo, posiblemente se reduzca el rendimiento si tenemos en cuenta un único hilo de ejecución. La técnica *AR-SMT* está basada en dos flujos de ejecución: flujo de ejecución activo, denominado *A-stream*, y el flujo de ejecución redundante, *R-stream*. El flujo de ejecución activo se corresponde con el programa original, por tanto, las instrucciones del mismo son ejecutadas y sus resultados utilizados en los diferentes estados del programa. Los resultados de cada instrucción son almacenados en un buffer *FIFO*, *Delay Buffer*. El flujo de ejecución redundante es ejecutado simultáneamente al flujo de ejecución activo, y los resultados que se obtienen en esta ejecución se comparan con los almacenados en el *Delay Buffer*. Si se produce alguna inconsistencia en la comparación se habrá detectado un error.

Otro método utilizado para detectar errores en los datos basado en la redundancia es *Data diversity*. *Data diversity* se basa principalmente en ejecutar dos programas diferentes, que tienen la misma funcionalidad, pero con diferentes conjuntos de datos, y comparar los resultados obtenidos. Esta técnica es capaz de detectar tanto errores permanentes como errores transitorios. Para poder implementar correctamente la técnica *Data diversity* se deben generar dos programas diferentes a partir de un mismo programa original. Una de las versiones es generada multiplicando todas las constantes y variables del programa por un factor *k*. Dependiendo del factor *k* el

programa original y el programa transformado podrían usar diferentes partes del hardware y propagar el fallo y sus efectos de diferentes maneras. Para determinar si se ha producido algún error en ejecución simplemente hay que comprobar si las salidas del programa modificado son *k* veces las del programa original. Las comprobaciones entre ambos programas pueden realizarse utilizando un tercer programa que se ejecute de manera concurrente y se encargue de las comparaciones. Otra opción es que el programa principal realice las comparaciones pertinentes después de finalizar las ejecuciones. Uno de las consideraciones a tener en cuenta es la necesidad de seleccionar un factor *k* que maximice la probabilidad de que los dos programas produzcan diferentes salidas provocadas por el efecto del mismo fallo. Además, hay que seleccionar el valor de *k* de manera que no se produzca un desbordamiento de la unidad funcional. El problema del desbordamiento puede ser resuelto realizando un ajuste de los datos. Una opción es utilizar datos de doble precisión para el programa duplicado. El problema es que hay un impacto sobre el rendimiento del sistema debido a que el tamaño de los datos es del doble. Otra opción es dividir los datos del programa original por el factor *k* en lugar de multiplicar los datos del programa transformado. El problema de esta opción es que se puede producir pérdida de precisión en la ejecución del programa. En este segundo caso únicamente los bits más significativos son los que se comparan, ya que estos no se ven afectados por la división.

El método propuesto en [NOh02d], llamado *ED⁴I* (*Error Detection by Diverse Data and Duplicated Instructions*) demuestra que el valor de *k* que maximiza la probabilidad de detectar un fallo es diferente para distintas unidades funcionales. Por tanto, dependiendo de las unidades funcionales que use el programa se deberá seleccionar un valor del factor *k* u otro. Esta técnica de endurecimiento requiere un tamaño de memoria mayor a dos veces el tamaño de la memoria del programa original, igualmente produce un impacto sobre el rendimiento del sistema que es también mayor a dos. *ED⁴I* únicamente puede ser aplicado a programas que contienen operaciones aritméticas, asignaciones, llamadas a subrutinas o estructuras que modifiquen el flujo de programa, no siendo aplicable a funciones que

realicen operaciones lógicas ,como por ejemplo, funciones booleanas u operaciones de rotación, ni a funciones logarítmicas y funciones exponenciales.

2.2.2.2. Aserciones ejecutables

La técnica propuesta en este apartado está principalmente basada en la ejecución de una serie de sentencias adicionales que se encargar de comprobar la integridad de los datos que forman parte de un programa. La eficacia de este método depende en gran medida de la aplicación que se va a ejecutar. Además, es necesario que el desarrollador conozca el sistema con el fin de identificar las aserciones ejecutables más adecuadas para el mismo. Utilizando aserciones ejecutables en principio se puede detectar cualquier error en los datos causado tanto por fallos software como por fallos hardware.

El método propuesto en [Hille00] describe de manera rigurosa como se deben de clasificar los datos que van a ser testeados. Los datos se clasifican en dos grandes categorías: señales continuas y señales discretas. A su vez estas dos categorías pueden dividirse en subcategorías. Para cada una de estas categorías se especifican una serie de restricciones como los valores límites o la tasa de variación límite que pueden tener los datos. Estas restricciones son posteriormente utilizadas en las aserciones ejecutables. La detección de errores se lleva cabo realizando una especie de test de restricciones, en donde la violación de una de estas restricciones supone la detección de un error.

En [Vint01] se propone un método basado en aserciones ejecutables, para aplicaciones de control, con un sistema de recuperación que implica un esfuerzo mínimo, al que se le ha denominado como *best effort recovey*. El estado de las variables y salidas es protegido mediante aserciones ejecutables para detectar errores usando las restricciones físicas de los objetos controlados. Se pueden detectar los siguientes casos de error:

- Si se detecta un estado incorrecto en la variable de entrada por medio de una aserción ejecutable durante una iteración del

algoritmo de control, se lleva a cabo una rutina de recuperación que utiliza una copia de seguridad del estado previo. En este caso no se realiza una recuperación real. Ya que el valor de la variable de entrada podría ser diferente de la utilizada en la anterior iteración. Como consecuencia el valor de las salidas podría ser ligeramente diferente del valor que tendrían si no se hubiera producido un error.

- Si se detecta que una salida es incorrecta se lleva a cabo una rutina de recuperación que consiste en recuperar el valor de las salidas en la iteración previa. La variable de estado es actualizado también con el valor que tenía en la iteración anterior. Igualmente el valor de las salidas podría ser ligeramente diferente del valor que tendrían si no se hubiera producido un error.

Se han realizado una serie de experimentos del método propuesto en [Vint01] utilizando un controlador de motor como caso de aplicación. Los resultados experimentales muestran que el 10.7% de los errores introducidos en los datos de la memoria caché y en los registros internos de la *CPU* producen un error en el sistema, utilizando el programa original sin endurecimiento software. Al realizar el endurecimiento software e implementar el método de recuperación *best effort recovey* se reduce el porcentaje de errores a un 3.2% demostrando que esta técnica es eficiente a la hora de reducir el número de errores en algoritmos de control.

2.3. Técnicas Hardware

Las técnicas Hardware se basan en la modificación del circuito para robustecerlo. En este caso la modificación es costosa ya que requiere un rediseño del circuito pero los resultados obtenidos suelen obtener una buena cobertura de errores ya que se puede acceder a cualquier parte de la arquitectura para modificarla.

Existe un inconveniente fundamental en la aplicación de las técnicas hardware dependiendo del ámbito ya que es necesario el conocimiento previo de la arquitectura del circuito y acceso a la misma para poder modificarla. Hoy en día existen numerosos circuitos y aplicaciones basados

en chips de los que se desconoce el detalle de su arquitectura y por tanto no son ni accesibles ni modificables. Claro ejemplo de ello son los COTS.

La técnica tradicional de endurecimiento hardware es TMR ("Triple Modular Redundancy") que está basada en triplicar el circuito o partes del mismo y votar para detectar fallos simples. Está técnica consigue mitigación total pero a conlleva un alto overhead en área utilizada. Para reducirlo se puede aplicar de forma parcial de forma que solo se endurezcan de forma selectiva las partes más críticas de los circuitos. Para ello previamente es necesario evaluar el circuito y detectar estas zonas críticas [Sanc16] [Prat06]. La mitigación conseguida se puede establecer como un compromiso entre área y cobertura de fallos.

Cuando el circuito estudiado es un microprocesador existen otro tipo de técnicas que se basan en añadir un módulo hardware adicional para observar el comportamiento del circuito bajo estudio. Ejemplo de ello son las propuestas de [Mich91] [Berg09] [Bens03]. En este caso la capacidad de detección depende de la capacidad de observación viable ya sea gracias a un módulo observador complejo o a la capacidad de acceder a los buses del sistema. Los observadores pueden ser módulos hardware realmente simples o conllevar una complejidad arquitectural tan elevada que los equipare en prestaciones al circuito que se desea observar. Idealmente la observación debería ser no intrusiva para no perturbar el funcionamiento normal del circuito.

Los observadores se suelen denominar en este ámbito watchdog processors y se suelen clasificar en activos y pasivos. Los activos son circuitos más complejos, realizan tareas más complicadas, y por tanto conllevan un overhead mayor en área [Mich91], [Berg09]. Los watchdog processors pasivos presentan un overhead de área menor ya que realizan tareas sencillas y su arquitectura no es compleja pero implican modificaciones en el software ejecutado por el microprocesador que se desea observar. Un watchdog processor pasivo se puede encargar de verificar si las firmas o aserciones insertadas en el software ejecutado por el microprocesador son

correctas. Normalmente al reducir su complejidad implican un aumento en la necesidad de memoria.

2.4. Técnicas Híbridas

Las técnicas híbridas son el resultado de combinar técnicas hardware y técnicas software. Estas técnicas tienen como objetivo principal alcanzar una tasa de detección de errores elevada, con el menor hardware posible y reduciendo el impacto sobre el rendimiento del sistema. Las técnicas híbridas consiguen este objetivo combinando las ventajas tanto de las técnicas software como de las técnicas hardware, mejorando la tasa de detección de errores y reduciendo el impacto sobre el rendimiento del sistema.

Un sistema híbrido típico está basado en la implementación de un módulo hardware externo, comúnmente denominado *watchdog processor,* que trabaja conjuntamente con una técnica de endurecimiento software. Una de las principales funciones del módulo hardware es realizar todas las comprobaciones de consistencia que se realizaban en la técnica de endurecimiento software. Al utilizar un hardware dedicado las comprobaciones de consistencia se realizan de una manera más eficiente y además se consigue reducir el tamaño del código endurecido, así como el tiempo de ejecución del programa. El módulo hardware debe observar la ejecución del microprocesador para obtener la información necesaria con el objetivo de implementar de una manera eficiente la técnica híbrida. Tradicionalmente el módulo hardware se conectaba al bus de la memoria caché o al bus de la memoria, tal y como se muestra en la Figura 6. La información en este punto de observación es proporcionada en la etapa de *fetch*, justo después de que la instrucción se haya leído de la memoria.

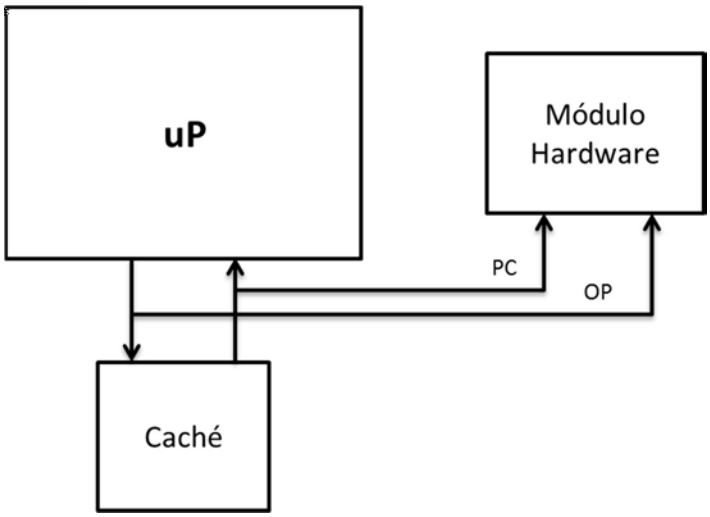

Figura 6: Utilización del bus de la memoria caché como punto de observación.

Otro punto de observación, que está empezando a ser utilizado en diferentes trabajos, es la interfaz de traza, tal y como se muestra en la Figura 7. A través de la interfaz de traza el módulo hardware obtiene la información más relevante de la ejecución justo después de que la instrucción haya sido ejecutada.

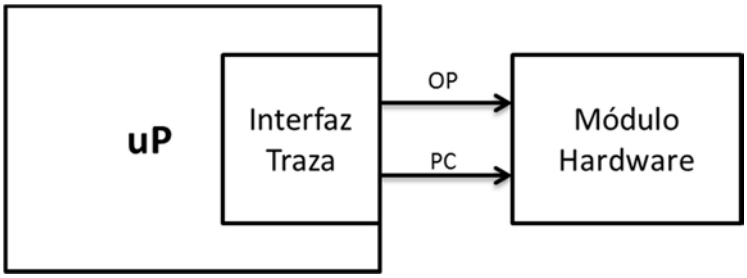

Figura 7: Utilización de la interfaz de traza como nuevo punto de observación.

La interfaz de traza forma parte de la unidad de depuración, y está principalmente pensada para llevar a cabo operaciones de depuración. Los microprocesadores más modernos utilizan este tipo de interfaces en sus operaciones de depuración. Gracias a que la unidad de depuración está inactiva durante una ejecución normal la interfaz de traza puede ser reutilizada para llevar a cabo una monitorización online de la ejecución del microprocesador. La interfaz de traza proporciona acceso a los recursos internos del microprocesador, sin necesidad de realizar modificaciones hardware y sin distorsionar la ejecución del mismo.

2.4.1. Utilización de la interfaz de memoria o bus

Las técnicas híbridas realizan la observación de la ejecución del microprocesador utilizando un único punto de observación, siendo este punto de observación el bus de la memoria caché. Gracias a la información que se obtiene a través del bus de la memoria caché se pueden detectar errores que afecten al flujo de programa. Para ello, se debe de comparar la información que se obtiene a través del punto de observación, con una serie de información de la ejecución que es calculada en tiempo de compilación. Todas estas comparaciones se realizan de una manera más eficiente en utilizando un hardware dedicado, obteniendo una mejora del rendimiento respecto a técnicas puramente software. En la literatura se puede encontrar una amplia variedad de técnicas híbridas que utilizan la interfaz de la memoria con punto de observación de la ejecución, a continuación se describen algunas de las más significativas.

En [Rodr11] se presenta una técnica híbrida basada en la utilización de firmas software en combinación con un módulo hardware que se encarga de monitorizar la ejecución. Mediante la implementación de dicha técnica híbrida se pretende detectar errores de tipo *SET* y *SEU*. En [Azam10] se muestran las principales limitaciones que tiene la implementación de una técnica puramente software basada en bloques básicos y en la utilización de firmas, describiendo cuales son los fallos que no podrían ser detectados.

La técnica híbrida propuesta en [Rodr11] propone dos conceptos principalmente con la finalidad de suplir las limitaciones de [Azam10] y detectar todos los tipos de errores. En primer lugar se proponen una serie de reglas de transformación del software cuya finalidad es controlar el módulo hardware. Por otro lado se introduce un módulo hardware encargado de monitorizar la ejecución del programa y de decodificar las instrucciones enviadas por las instrucciones software añadidas. La técnica software propuesta se basa en añadir instrucciones extra con la finalidad de proteger el sistema frente a saltos incorrectos dentro del mismo bloque básico y frente a saltos incorrectos hacia el inicio de otro bloque básico. Con la finalidad de detectar errores que produzcan un salto incorrecto al inicio de otro bloque

básico, el programa es dividido en bloques básicos, a los que se les asigna dos identificadores *CFID* y *BID*, implementando una técnica muy similar a la técnica *ECCA*, descrita en el apartado 2.2.1.1. El identificador *BID* (*Block IDentifier*) representa a cada bloque con un número primo único mientras que el identificador *CFID* (*Control flow IDentifier*) representa el flujo de programa, almacenando el producto de los identificadores *BID* de los siguientes bloques básicos. Debido a que los identificadores *BID* únicamente están compuestos de números primos el resto de dividir el identificador *CFID* por el identificador *BID* siempre es cero, habiéndose producido un error en caso contrario. Los identificadores *CFID* son almacenados en una cola de dos elementos. La cola es inicializada con el identificador del primer bloque. Cada vez que se entra en un nuevo bloque el *CFID* del siguiente bloque es almacenado en la cola. Cada vez que se sale de un bloque, se coge el identificador de la cola y se divide por el identificador *BID*. Se detecta un error cuando ocurre alguna de las siguientes situaciones: el resto de la división es distinto de cero, *overflow* o *underflow* en los elementos de la cola. El manejo de la cola es una tarea muy pesada para ser realizada mediante software y podría resultar en un incremento importante de memoria produciendo una pérdida de rendimiento. Por esta razón, el control de todo el sistema relacionado con la cola y las divisiones son realizados por el módulo hardware, siendo controlado desde el software. Por otro lado, para detectar errores dentro de un mismo bloque básico se calcula la firma de cada bloque utilizando la función lógica *XOR*. Las firmas son calculadas en tiempo de compilación y se envían al módulo hardware, añadiendo instrucciones adicionales en el software. Cada vez que se finaliza un bloque básico el software envía la firma pre-calculada al módulo hardware, con la finalidad de que pueda comparar la firma online con la offline. Es el módulo hardware el que se encarga de calcular la firma online y de notificar que se ha producido un error cuando las firmas difieren.

El módulo hardware que se propone en [Rodr11] combina las funciones de decodificación y monitorización de la ejecución. Mediante la monitorización de la ejecución se consigue detectar saltos incorrectos a direcciones de memoria no utilizadas y bucles infinitos, mientras que con la

función de decodificación se observan los datos y señales de lectura/escritura transferidos entre el procesador y la memoria con el objetivo de detectar las instrucciones añadidas en el software que indican o controlan el funcionamiento del módulo hardware. El módulo hardware tiene que ser capaz de acceder a los buses de la memoria, implementar una cola de dos elementos, un módulo decodificador y realizar operaciones de división.

Con la finalidad de comprobar la capacidad para detectar errores de la técnica propuesta, los autores han realizado una campaña de inyección de fallos en simulación utilizando el microprocesador *MIPS* como caso de estudio. Se inyectaron un total de 50,000 fallos *SEUs* y *SETs* en todas las señales no protegidas del microprocesador, utilizando la herramienta *Modelsim.* Los resultados que obtienen muestran una elevada eficiencia en la detección de errores, ya que se alcanza una cobertura a fallos del 100%, con un incremento del tiempo de ejecución que varía entre el 133% y el 153%.

En [Rodr13] se propone la técnica híbrida *HETA* (*Hybrid Error-Detection Technique Using Assertions*). *HETA* es una técnica híbrida basada en la utilización de un módulo hardware que monitoriza la ejecución, en combinación con una técnica de endurecimiento software basada en la utilización de aserciones. El principal objetivo que busca la técnica *HETA* es detectar errores en el flujo de programa. Para alcanzar este objetivo se combina una técnica software con un módulo hardware. La comunicación entre la técnica software y el módulo hardware se realiza mediante la utilización de instrucciones estratégicamente insertadas en el código, las cuales pueden ser interpretadas por el módulo hardware. La técnica de endurecimiento software que proponen los autores está originalmente basada en *CEDA,* descrita en el apartado 2.2.1.3. La técnica *CEDA* utiliza firmas para detectar errores que afecten al flujo de programa. Esta técnica es capaz de detectar fallos que no sigan el gráfico del flujo de programa, sin embargo, no es capaz de detectar saltos incorrectos pero que son legales de acuerdo con el flujo de programa. Con la finalidad de detectar

estos errores se utiliza un módulo hardware que monitoriza la ejecución, como el propuesto en [Rodr11].

La técnica *HETA* genera las firmas basándose en el gráfico del programa que representa el flujo del mismo. El gráfico de programa es divido en nodos y a cada nodo se le asignan tres firmas, *NIS* (*Node Ingress Signature*), *NS* (*Node Signature*), *NES* (*Node Exit Signature*). La firma *NIS* se corresponde con el valor esperado de la firma online al ingresar en el nodo, mientras que la firma *NS* es el valor esperado de la firma en cualquier punto interior del nodo y la firma *NES* es el valor esperado de la firma a la salida del nodo. Para poder actualizar las firmas de manera online y realizar una monitorización de la ejecución se insertan instrucciones en el código del programa. Cuando la ejecución alcanza un nuevo nodo la firma online es actualizada con el valor de la firma *NIS;* durante la ejecución del nodo la firma se actualiza con el valor que tenga *NS*, mientras que cuando se finaliza la ejecución del nodo la firma es actualizada con el valor que tenga *NES*. Observando las firmas *NES* y *NIS* se pueden detectar en el flujo de programa causado por un salto entre diferentes nodos. Mientras que mediante la firma *NS* se pueden detectar errores en el flujo que se produzcan en el mismo nodo. En ciertas partes del código se introducen comprobaciones, para detectar diferencia entre las firmas, utilizando instrucciones. Estas instrucciones almacenan el valor de la firma online en una memoria de manera que pueda ser identificadas por el modulo hardware.

Uno de los puntos a tener en cuenta es la manera en que se deben actualizar las firmas. A la firma *NS* se le debe asignar un valor que pueda ser calculado por el módulo hardware, además debe tener un valor único. Para conseguir estas características la firma *NS* es calculada realizando la función lógica *XOR* de todas las instrucciones que forman parte del nodo más la dirección de memoria de la primera instrucción. Para calcular el valor de las firmas *NIS* y *NES* se divide el cálculo en dos partes, la parte baja y la parte alta. Cada parte se calcula de diferente manera y es utilizada para diferentes objetivos. La parte alta es utilizada para identificar la red de nodos del programa, mientras que la parte baja se usa para identificar los nodos

dentro de una red. Esta técnica al igual que ocurre en la técnica *CEDA* no es capaz de detectar saltos incorrectos pero legales de acuerdo con el gráfico del programa. Para detectar este tipo de errores los autores utilizan una técnica denominada *Inverted Branches*. Sin embargo, *HETA* es capaz de detectar errores que se produzcan dentro de un mismo nodo, cosa que la técnica *CEDA* no es capaz.

El módulo hardware que se utiliza en la técnica *HETA* tiene dos funcionalidades principalmente, monitorizar la ejecución y decodificar las instrucciones para poder recibir las indicaciones enviadas por el software. Realizando la monitorización de la ejecución se busca detectar saltos incorrectos a direcciones de memoria no utilizadas y bucles infinitos. El decodificador que se implementa en el módulo hardware observa los datos y señales de lectura/escritura transferidos entre el procesador y la memoria con el objetivo de detectar las instrucciones añadidas en el software que indican al módulo hardware que debe realizar actualizaciones en las firmas o una comparación de las mismas. El decodificador lee los buses buscando dos instrucciones. La primera de ellas, denominada como *ResetXOR* por los autores, indica que debe de reiniciarse la firma online, mientras que la segunda instrucción, denominada *CheckXOR*, indica que hay que comparar las firmas para comprobar que el valor de la firma online es correcto. Para que el módulo hardware pueda implementar ambas funcionalidades, además de tener acceso a los buses de memoria, debe de implementar un operador *XOR*, un módulo decodificador y un módulo de error, que es activado cuando se ha producido un error durante la ejecución del programa.

La técnica *HETA* no es capaz de detectar la totalidad de los errores que se producen en el flujo de programa, ya que no es capaz de detectar errores en el flujo que no violen el gráfico del flujo de programa. Además, no se implementa ningún mecanismo para detectar errores que se produzcan en los datos del programa. Por esta razón, los autores proponen combinar *HETA* con dos técnicas de endurecimiento software que hagan frente a estos tipos de errores. La primera de las técnicas que se implementa es

denominada *Inverted branches* [Azam11b] y es capaz de detectar errores en la condición de los saltos condicionales. Los errores que se dan en la condición de los saltos condicionales afectan a la transición entre bloques básicos y son difíciles de detectar, ya que la condición evaluada puede dar lugar a la ejecución de dos instrucciones diferentes dentro del flujo del programa pero dependiendo de los datos utilizados únicamente uno de ellos es correcto. Teniendo esto en cuenta un fallo puede provocar un cambio en el dato evaluado que ocasione un cambio en la evaluación de la condición. Si al evaluar la condición se realiza un salto permitido pero erróneo, será indetectable.

La técnica inverted branches se basa principalmente en replicar la instrucción de salto. Para ello hay que tener en cuenta que el salto puede ser tomado o no. Para los casos en los que no se toma el salto bastaría con replicar la instrucción justo después de la instrucción original, y utilizando como dirección de salto el inicio de una subrutina de tratamiento de errores. Como el salto de la instrucción original no ha sido tomado, tampoco debería llevarse a cabo en la instrucción replicada, de manera que si se produce el salto en la instrucción replicada se habrá detectado un error. En el caso en el que el salto es tomado la instrucción replicada es insertada en el destino del salto, de manera que sea ejecutada después de llevar a cabo el salto. En este caso la instrucción de salto replicada también debería ser tomada. Para mantener el flujo de programa se invierte la condición de salto y el destino del salto apunta al inicio de una rutina de error. En este caso al haber invertido la condición el salto no debería ser tomado, en caso contrario se habría producido un error.

La segunda técnica de endurecimiento software que se implementa es denominada *Variables.* La técnica *Variables* replica las variables que utiliza el programa y las operaciones de escritura realizadas sobre dichas variables. Además. Se realizan comprobaciones de consistencia entre las variables y sus réplicas cada vez que se realiza una operación de lectura o cuando se ejecuta una instrucción de salto. Realizando estas comprobaciones de consistencia se consigue asegurar que los datos que son almacenados y

leídos de la memoria son correctos, igualmente se asegura que los datos utilizados en las instrucciones de salto están exentos de error. Las transformaciones necesarias para implementar la técnica suponen duplicar los datos a almacenas, el número de registros y las direcciones de memoria. Por tanto, esta técnica está limitada a la cantidad de recursos que hay disponibles y habrá que buscar una solución de compromiso en cada caso particular, llegando a endurecer mediante duplicación las zonas más críticas de la ejecución del programa.

Con la finalidad de determinar la capacidad de la técnica *HETA* a la hora de detectar errores, los autores han realizado una campaña de inyección de fallos en simulación utilizando el microprocesador *MIPS* como caso de estudio. Los experimentos consistieron en la inyección de fallos tipo *SEU* y *SET* utilizando la herramienta *Modelsim*. Los resultados muestran una elevada eficiencia a la hora de detectar errores, alcanzando una cobertura a fallos del 100%. Sin embargo, se produce un impacto sobre el rendimiento del sistema en torno al 11% y un incremento en el área del 55%.

Una técnica híbrida similar es presentada en [Rodr12]. La técnica híbrida propuesta por los autores combina la utilización de un módulo hardware con dos técnicas de endurecimiento software. El módulo hardware, al que los autores han denominado *Online control-flow checker module* (*OCFCM*), monitoriza la ejecución del programa realizando la observación entre el microprocesador y la memoria de programa. Para llevar a cabo la monitorización del programa el *OCFCM* comprueba los accesos a memoria y las instrucciones que produzcan un cambio en el flujo de programa. Para poder alcanzar una cobertura a fallos importante es necesario implementar el módulo *OCFCM* en combinación con técnicas de endurecimiento software; en este caso se han seleccionado las mismas que se utilizan en [Rodr13]: *Inverted Branches* y *Variables*. La técnica *Inverted branches* detecta errores que afecten a los bits de condición en saltos condicionales, mientras que la técnica *Variables* protege el sistema frente a errores que afecten a los datos de programa.

El principal objetivo del módulo *OCFCM* es detectar errores que provoquen saltos incorrectos en el flujo de programa. Para conseguir este objetivo los autores han combinado muchas de las técnicas hardware no intrusivas existentes en la literatura. El módulo *OCFCM* comprueba si el procesador está accediendo a áreas de memoria correctas, la consistencia de determinadas variables, determina si el flujo de programa es correcto, así como la evolución del contador del programa durante la ejecución del programa. Para llevar a cabo todas estas funciones el módulo *OCFCM* almacena cierta información de la ejecución calculada en tiempo de compilación y además observa la ejecución a través de los buses de la memoria. El módulo *OCFCM* es módulo dependiente de la aplicación a ejecutar, y almacena información de las zonas de memoria que son permitidas para la aplicación dada. Gracias a esta información, se puede determinar si los accesos a memoria son correctos, tanto para los datos como para las instrucciones. Además, se comprueba que el contador de programa sea el correcto durante la ejecución del programa. Comprobando la evolución del contador de programa, el módulo *OCFCM* puede determinar si la ejecución se ha quedado apuntando a la misma dirección de programa, comprobando el número de ciclos de reloj que se utilizan para ejecutar dicha instrucción. Este tipo de errores son importantes ya que no pueden ser detectados por técnicas puramente software. Adicionalmente, el módulo *OCFCM* tiene la capacidad de comprobar en tiempo de ejecución que los saltos realizados son correctos. Para ello comprueba el valor del contador de programa y determina si el programa está siguiendo un flujo de programa permitido dentro del diagrama de flujo. Con la finalidad de detectar inconsistencias en el flujo de programa el módulo *OCFCM* debe buscar y decodificador cada una de las instrucciones ejecutadas por el programa, de manera que pueda identificar las instrucciones de salto y las direcciones de destino del salto.

Con la finalidad de determinar la eficiencia de la técnica propuesta, los autores han realizado una campaña de inyección de fallos utilizando la *FPGA ProASIC3* de *Actel.* Los experimentos han consistido en la inyección de fallos *SET* y *SEU*, uno por simulación, alcanzado un total de 40,000 fallos por

aplicación caso de estudio y utilizando un total de 6 aplicaciones diferentes. Los resultados que se obtienen muestran que la técnica híbrida propuesta por los autores presenta una elevada capacidad para detectar errores, alcanzando una detección de la totalidad de los errores observados en los experimentos.

2.4.2. Utilización de la interfaz de depuración

El uso de la interfaz de traza, presente en la unidad de depuración, como punto de observación de la ejecución está creciendo en los últimos años. A través de la interfaz de traza se puede acceder a la información más relevante de la ejecución, como por ejemplo el contador de programa y el código de operación. Durante operaciones de depuración el microprocesador transfiere, a través de la interfaz de traza, la información más importante de la ejecución a un buffer encargado de almacenarla, denominado buffer de traza. Gracias a que la unidad de depuración está inoperante durante una ejecución normal, se puede tener acceso a la interfaz de traza de una manera no intrusiva, y por tanto, obtener acceso a información de la ejecución sin influir en la misma. La interfaz de traza proporciona la información de la ejecución en la etapa de ejecución del pipeline. Por tanto, utilizando la interfaz de traza como punto de observación, se podrá detectar un error que se produzca o que acabe afectando a cualquiera de las etapas del pipeline anteriores a la de ejecución. Otra de las ventajas que presenta la utilización de la interfaz de traza es que se está reutilizando un hardware existente, de manera que el coste de utilizarlo como nuevo punto de observación es cero, además de no tener que realizar modificación alguna al hardware para su utilización.

El trabajo propuesto en [Gros10] propone la utilización de la interfaz de traza del microprocesador *LEON3* [Gais01] como punto de observación de la ejecución. Los autores desarrollan una técnica para la detección de errores, cuya estrategia de detección está basada en la duplicación de las tareas ejecutadas en un sistema basado en un microprocesador. Para llevar a cabo la duplicación de la ejecución se puede utilizar un mismo microprocesador y replicar la ejecución o utilizar un microprocesador que

tenga varios núcleos que ejecuten el mismo programa en diferentes instantes de tiempo. Durante cada ejecución se observan las señales enviadas por el microprocesador a través de la interfaz de traza y se generan firmas en tiempo de ejecución con parte de la información que almacenan dichas señales. Las firmas generadas durante la ejecución son comparadas con la finalidad de detectar posibles errores que se hayan producido en el programa, y en el caso que difieran se activará un proceso de recuperación, que puede consistir en la repetición de la tarea donde se ha producido el error. La firmas son calculadas a partir de la información que haya disponible en la interfaz de traza, típicamente el contador de programa, el código de operación y una serie de flags del microprocesador. Los resultados experimentales muestran que es importante seleccionar correctamente la información a partir de la que se genera las firmas para obtener una buena cobertura a fallos. Una de las principales ventajas de la técnica propuesta en [Gros10] respecto al estado del arte, es que alcanza una elevada eficiencia a la hora de detectar errores y con una latencia mínima, alcanzada gracias a la reutilización de las infraestructuras de depuración disponibles. Los autores introducen un hardware externo, denominado *Debug controller*, encargado de generar las firmas, realizar las comparaciones, etc… No se requiere ninguna modificación en los microprocesadores utilizados, y el impacto sobre el rendimiento general del sistema se reduce a la latencia añadida al implementar la redundancia. En el trabajo presentado en [Gros10] los autores deciden implementar la técnica en un sistema multiprocesador en lugar de duplicar la ejecución. Considerando que los dos procesadores utilizados tienen la misma arquitectura y ejecutan el mismo programa con datos duplicados, se espera que realicen exactamente las mismas operaciones y que los resultados obtenidos sean equivalentes. Cuando un fallo afecta a uno de los procesadores es detectado por el *Debug controller* comparando las firmas que se han generado. El módulo *Debug controller* debe realizar las siguientes tareas:

- Monitorizar la ejecución del programa observando la información que se proporciona a través de la interfaz de traza.

- Realizar el cálculo de la firma online, comprimiendo la información obtenida de la interfaz de traza.
- Comparar las firmas calculadas en ambas ejecuciones.
- Notificar la detección de un fallo en el sistema, habilitando la ejecución de una rutina de recuperación.

Además de todas estas tareas, el módulo *Debug controller* reserva una zona de memoria, dentro del mismo módulo, para almacenar las direcciones de inicio y duraciones de todas las zonas de la ejecución que van a ser replicadas. De esta manera se consiguen detectar pérdidas de secuencia en la ejecución. Las firmas son calculadas usando un *MISR* (*Multiple Input Signature Register*) a partir de los datos proporcionados por la interfaz de traza. La primera copia de la firma que se genera es almacenada para poder ser comparada posteriormente con la segunda copia.

Los autores han realizado una serie de experimentos utilizando el microprocesador *LEON3* como caso de estudio, con la finalidad de realizar un análisis de la técnica propuesta. Se han utilizado dos herramientas diferentes para llevar a cabo dos experimentos independientes. Por un lado, se han realizado experimentos de inyección de fallos transitorios, tipo *SEU*, en los que se ha utilizado la herramienta *Modelsim.* Además, se han llevado a cabo experimentos de inyección de fallos permanentes, en los que se ha utilizado la herramienta *Synopsys Tetramax*. Se han utilizado un total de cuatro aplicaciones software para el desarrollo de ambos experimentos. En los primeros experimentos se ha inyectado un fallo transitorio tipo *SEU* por simulación. Los fallos han sido inyectados en los registros del núcleo del procesador de una manera aleatoria. En los segundos experimentos se ha inyectado un fallo permanente en el núcleo de uno de los microprocesadores. Los resultados obtenidos en los experimentos de inyección de fallos permanentes no alcanzan una cobertura a fallos elevada. Esta baja cobertura se debe a que las aplicaciones ejecutadas no han sido desarrollada teniendo en cuenta el test que se iba a realizar. Además hay que tener en cuenta que un dispositivo que presenta un error

permanente va a reproducir la misma firma incorrecta cada vez que se ejecute el programa. Sin embargo, en los experimentos de fallos transitorios se han alcanzado unos niveles de cobertura a fallos elevados. La mayoría de los fallos que provocan un resultado incorrecto o que alteran el flujo de programa son detectados, alcanzando una detección superior al 99.4%. Por tanto, se puede concluir que utilizando la interfaz de traza como nuevo punto de observación se pueden alcanzar unos resultados satisfactorios.

En [Port12] se extiende el trabajo propuesto en [Gros10] proponiendo un caso general. Se propone un método para detectar errores permanentes y transitorios que se puedan producir en un microprocesador utilizando la interfaz de traza como punto de observación de la ejecución. Las tareas que son consideradas críticas para la aplicación son replicadas ya sea utilizando redundancia temporal y/o replicando el hardware. El problema de seleccionar cuales son las tareas a replicar depende de las especificaciones del sistema. Hay que buscar una solución de compromiso entre el nivel de protección y el coste que lleva asociado; las diferentes soluciones han sido ampliamente debatidas en la literatura. Como se ha comentado anteriormente en [Port12] se propone la utilización de la infraestructura de depuración como punto de observación de la ejecución. Existen dos alternativas para capturar la información que es almacenada en la traza. Por un lado, se puede interceptar la información que es enviada por el procesador hacia la interfaz de traza, o simplemente leer la información que ya está almacenada en el buffer de traza. La primera solución proporciona mayor cobertura a fallos y minimiza la latencia, sin embargo, se requiere tener acceso a la interfaz de traza. La segunda opción es usada cuando no se tiene acceso a la interfaz de traza. En este caso la latencia es mayor y la cobertura a fallos podría ser menor dependiendo de del tamaño del buffer de traza y de la política de acceso.

Para implementar el método, los autores han desarrollado un módulo hardware personalizable, al que han denominado *CPU_Checker*, encargado de observar las tareas que son replicadas. Este módulo puede integrarse en un sistema basado en microprocesador sin la necesidad de

realizar modificaciones en el núcleo del procesador, o conectarse a un puerto de traza externo. El *CPU_Checker* calcula una firma para cada una de las tareas replicadas utilizando la información obtenida de la infraestructura de depuración. Cada vez que finaliza la ejecución de una tarea crítica se comparan las firmas de ambas ejecuciones con el objetivo de comprobar si se ha producido un error en la ejecución de dicha firma. La arquitectura del *CPU_Checker* está dividida en los siguientes bloques: memoria, control y uno o más *CPU Observers*. La memoria almacena toda la información necesaria para implementar el método. Es dividida en cuatro bloques para almacenar las direcciones de inicio y de fin de cada una de las tareas críticas, el tiempo máximo permitido para la ejecución de una tarea y la firma calculada en la primera ejecución de la tarea. El control se encarga de coordinar el resto de bloques para poder detectar los errores que se produzcan durante la ejecución de una tarea crítica. Cada bloque *CPU Observer* se encarga de monitorizar la ejecución de tareas críticas. El número de bloques *CPU Observer* que componen el *CPU_Checker* depende del número de réplicas de tareas críticas que se ejecutan simultáneamente. Cada *CPU Observer* se compone de los siguientes elementos:

- Módulo MISR: módulo encargado de calcular la firma de una tarea crítica utilizando los datos obtenidos de la interfaz de traza.
- Watchdog timer: módulo encargado de comprobar si las tareas finalizan dentro del tiempo permitido.
- Observer's Controller: se encarga de controlar los otros dos elementos que conforman el CPU observer, es decir, controla el funcionamiento del Módulo MISR y el Watchdog timer.

El primer paso que hay tomar es configurar el *CPU_Checker*. Para ello se debe de almacenar en la memoria las direcciones de inicio y de fin de cada una de las tareas críticas, así como el tiempo máximo de ejecución de cada una de las tareas. Una vez que se ha llevado a cabo la configuración se puede iniciar la ejecución en uno o varios procesadores, dependiendo de

la solución por la que se haya optado. Cada vez que inicie la ejecución de una tarea crítica, el módulo *MISR* inicia el cálculo de una firma para dicha tarea. Cuando finaliza la primera ejecución de la tarea la firma es almacenada en la memoria. Una vez que se haya finalizado la ejecución de las réplicas de una tarea se comparan las firmas, activando una señal de error en caso que no sean iguales. Hay principalmente tres posibles casos de error:

- las instrucciones ejecutadas por una tarea y sus réplicas son diferentes, lo que acaba produciendo que las firmas sean diferentes.
- El orden de ejecución de las réplicas de una tarea se ve alterado. Se considera un error en sistemas donde el orden de ejecución es relevante.
- Una de las dos réplicas no finaliza la ejecución.

Cuando no se detecta ningún error en una tarea dada, el microprocesador sigue ejecutando otras tareas. En el caso que si se detecte un error se ejecutaran una serie de procesos de recuperación.

El *CPU_Checker* puede ser implementado en diferentes sistemas con arquitecturas distintas y combinados con diferentes técnicas de endurecimiento. A continuación se muestran las diferentes opciones en las que puede ser implementado:

- Redundancia temporal: la redundancia temporal es una técnica basada en la repetición de la ejecución de las tareas críticas en diferentes instantes de tiempo. Los autores implementan el método propuesto comprobando la ejecución de las tareas repetidas con la finalidad de detectar fallos transitorios. Utilizando redundancia temporal un fallo podría no solo ser detectado sino incluso corregido en el caso que se triplique la ejecución de tareas críticas. En este caso, únicamente hay un módulo *CPU Observer* en el *CPU_Checker* ya que únicamente se ejecuta una tarea cada vez.

- Redundancia Hardware: otra posibilidad para implementar el método propuesto es llevar a cabo la ejecución de las tareas críticas en uno o más procesadores idénticos. El *CPU_Checker* tiene la misión de comparar ambas ejecuciones y advertir a los procesadores en el caso que se produzca un error. La principal ventaja de esta opción de implementación es que es una solución no intrusiva; no se altera la ejecución del microprocesador y no se requiere esfuerzo computacional extra. Como en el caso anterior si se utiliza un tercer microprocesador, los errores simples pueden ser corregidos. Además se pueden detectar fallos permanentes en uno de los microprocesadores y llevar a cabo acciones de recuperación, como por ejemplo reemplazar el módulo que falla.

- Sistema multiprocesador: en un sistema multiprocesador se puede alcanzar máxima eficiencia ya que diferentes procesadores pueden ejecutar diferentes tareas simultáneamente. En este tipo de arquitecturas se puede utilizar tanto la redundancia temporal como la redundancia hardware beneficiándose de los procesadores disponibles. Con el objetivo de implementar el método propuesto varios procesadores pueden ejecutar una misma tarea crítica, siendo el *CPU_Checker* el encargado de comparar ambas ejecuciones. El *CPU_Checker* obtiene la información de la ejecución a través de la interfaz de traza de cada uno de los microprocesadores, con esta información genera las firmas para cada una de las tareas y las compara al finalizar ambas ejecuciones.

2.5. Resumen y conclusiones

En este capítulo se ha resumido el estado de la técnica en el ámbito de la detección de errores en arquitecturas microprocesadores. En este campo cabe destacar la utilización de diversos subconjuntos de técnicas que se han detallado a lo largo del capítulo. La primera división que tradicionalmente se realiza es la de abordar de qué forma se va a implementar el endurecimiento de la arquitectura si es vía hardware o software.

En la vía hardware el circuito se modifica para hacerlo más robusto. Dentro de la vía hardware la técnica más comúnmente utilizada es TMR. Estás técnicas que modifican la arquitectura consiguen coberturas muy altas a cambio de un alto coste en área y un gran esfuerzo de diseño con la necesidad intrínseca de la utilización de personal específicamente cualificado para realizar esta tarea. Además presentan el inconveniente de que es necesario conocer al detalle la arquitectura que se desea modificar y esto hoy en día no es viable para multitud de circuitos. Existe otra aproximación que se basa en observar el circuito a través de algún interfaz disponible. Muchos trabajos han elegido el interfaz con las memorias porque es el más sencillo de observar y a él han conectado módulos hardware para observar la ejecución y poder observar los posibles errores. Los módulos hardware conectados se suelen denominar watchdog processors y pueden ser módulos muy sencillos que realicen tareas triviales hasta procesadores tan complejos que serían equivalentes al que se está evaluando.

Las técnicas software realizan modificaciones sobre el software que ejecuta el microprocesador. En este caso la modificación es más accesible y sencilla porque se trata de modificar un código. Sin embargo la modificación del código implica también sus inconvenientes, al robustecer el código es necesario introducir instrucciones así como estructuras de datos adicionales que alargan el tiempo de ejecución, aumentan la necesidad de memoria y disminuyen las prestaciones. Otro problema intrínseco de este tipo de técnicas es que solo se puede robustecer aquello que es accesible desde el modelo de programación y este no accede a la arquitectura completa. Por lo tanto, las técnicas software presentan per se un hándicap en la cobertura que será palpable dependiendo del tipo de software que se utilice para testar el circuito. Dentro de las técnicas software tradicionalmente existe una separación entre técnicas que estudian el control de flujo y técnicas que tratan los datos. El control de flujo busca detectar errores en el flujo de programa, es decir saltos a ubicaciones en memoria de programa incorrectas o saltos permitidos dentro del flujo pero tomados en instantes inadecuados. En este capítulo se han detallado una extensa muestra de técnicas que abordan el control de flujo de programa,

de las más utilizadas son las firmas y las aserciones. Para el control de flujo suele utilizarse la división del código en bloques básicos (BB) que son conjuntos de instrucciones secuenciales, que no tienen instrucciones de salto. Las técnicas basadas en firmas realizan una o varias firmas asociadas a los BB que se calculan y almacenan en tiempo de compilación y se recalculan durante la ejecución para su comparación. Es evidente que la utilización de firmas conlleva un aumento en la necesidad de memoria y a su vez la necesidad de modificar el software para generar las firmas y compararlas. Las técnicas basadas en aserciones introducen sentencias especiales dentro del código para realizar comprobaciones, las aserciones son menos sistemáticas y dependen más del código a tratar y de la pericia del programador para insertar las aserciones en los lugares óptimos.

Respecto a las técnicas de datos las más utilizadas son las basadas en duplicación. La duplicación se puede realizar con mayor o menor granularidad desde las instrucciones hasta la propia duplicación del programa a ejecutar. Disminuir la frecuencia de la duplicación permite la reducción de la penalización en tiempo de ejecución, pero sin embargo aumenta la latencia en la detección de los errores. Existen técnicas que buscan el compromiso entre ambos factores.

Además de técnicas hardware y software también se han presentado en la literatura técnicas híbridas que tratan de aunar las ventajas de ambos tipos de técnicas. Las técnicas presentadas en este ámbito son muy variadas y dependen de la arquitectura utilizada y de las conexiones disponibles para la ubicación de módulos hardware que observen el comportamiento. La mayoría de las técnicas utilizan técnicas software para el control de los errores en los datos ya que el acceso desde el software a las estructuras de datos es más sencillo que desde el hardware.

A partir del estado de la técnica detallado en este capítulo se desarrollarán los siguientes capítulos objeto del trabajo desarrollado en esta tesis. En el trabajo realizado en esta tesis se han propuesto nuevas técnicas y avances en las descritas en este capítulo que modifican y enriquecen el estado de la técnica actual.

CAPÍTULO 3

TÉCNICAS DE DETECCIÓN DE ERRORES DE CONTROL DE FLUJO

3. Técnicas de detección de errores de control de flujo

En este capítulo se describen de manera detallada las técnicas para detectar errores en microprocesadores que han sido desarrolladas a lo largo de la tesis doctoral. Estas técnicas están basadas principalmente en la detección de errores que afectan al flujo de instrucciones durante la ejecución de un programa, comúnmente conocidas como técnicas de *Control Flow Checking* (*CFC*).

El objetivo de estas técnicas es mejorar el compromiso entre la tolerancia a fallos y el impacto sobre las prestaciones del sistema, buscando mejorar este compromiso respecto a las técnicas existentes. Por otra parte, un objetivo fundamental es desarrollar técnicas no intrusivas, es decir, que no afecten a la ejecución normal del microprocesador, así como que su implementación pueda llevarse a cabo sin tener que realizar modificaciones en el diseño original del mismo. Este último requisito es fundamental para facilitar su uso, dado que el diseño y fabricación de un microprocesador es una tarea costosa y compleja.

Para este propósito, todas las técnicas que se proponen en este capítulo están pensadas para ser implementadas mediante un módulo hardware externo, el cual observa el flujo de instrucciones que ejecuta el microprocesador desde una interfaz apropiada. Esta solución arquitectural se beneficia de su implementación en hardware para mejorar la eficiencia y no es intrusiva, puesto que el hardware adicional es externo.

Todas las técnicas que se describen en este capítulo son aportación original de esta tesis doctoral.

3.1. Introducción

Las técnicas de *CFC* se centran principalmente en la detección de errores que afectan al flujo de ejecución de un programa, en inglés denominados *Control Flow Errors* (*CFEs*). Un *CFE* produce una variación en el flujo de programa, de manera que el microprocesador no ejecuta las

mismas instrucciones que se deberían ejecutar en ausencia de error. Las técnicas de *CFC* han sido ampliamente estudiadas por la comunidad científica [Ohsh02], [Vemu06], [Wilk88], [Alkh99], [Chen05], [Mich91], [Made91], [Bens03], debido a que un porcentaje muy elevado de los errores que se producen en un sistema basado en microprocesador son errores que afectan al flujo de programa [Ohls92]. Se han conseguido alcanzar soluciones que obtienen porcentajes de detección cercanos al 100% de los errores de control de flujo, aunque ninguna de las técnicas existentes es completamente efectiva [Azam11a]. Por otra parte, todas estas técnicas presentan un problema común: el fuerte impacto sobre el rendimiento del sistema que produce su implementación.

Para llevar a cabo el control del flujo de instrucciones es necesario tener cierta información de la ejecución para poder realizar una comparación *online* con las instrucciones que son ejecutadas por el microprocesador. Una de las soluciones más utilizadas es almacenar información sobre la ejecución, que es calculada en tiempo de compilación, y usada posteriormente para llevar a cabo las comparaciones [Mich91], [Berg09]. Esto supone un importante incremento en el tamaño de los recursos de memoria necesarios para la implementación del sistema, así como de la complejidad de los mecanismos para la comprobación, que redundan en una degradación del rendimiento del sistema. Si además las comparaciones son realizadas mediante software el impacto sobre el rendimiento es todavía mayor.

Otra de las posibles soluciones consiste en duplicar la ejecución, de manera que se comparan los resultados obtenidos en ambas ejecuciones [Gros10]. Habitualmente se compacta la información de flujo calculando una firma que posteriormente se compara entre las dos ejecuciones. Igualmente, en este segundo caso el impacto sobre el rendimiento del sistema es elevado, ya que el tiempo de ejecución se ve aumentado por más del doble, produciendo una pérdida de prestaciones.

Aunque las técnicas de *CFC* no proporcionan una solución completa frente a los errores que se producen en un microprocesador, diferentes

estudios demuestran que son este tipo de errores los más abundantes en sistemas basados en microprocesador [Ohls92]. Teniendo en cuenta este dato y el fuerte impacto sobre las prestaciones de las técnicas de *CFC* existentes, se ha decidido realizar, a lo largo del desarrollo de la tesis doctoral, el diseño de técnicas de *CFC* que tengan como objetivo alcanzar una detección de errores elevada tratando de reducir la necesidad de almacenar información en tiempo de compilación o de llevar a cabo duplicaciones en la ejecución. Para conseguir este objetivo, la primera opción que se ha explorado es aprovechar información que está implícita en el propio flujo de instrucciones, como el código de operación y la dirección de cada instrucción. Durante la ejecución de un programa correcto, las instrucciones guardan cierta relación entre sí, la cual da lugar a ciertas condiciones que se pueden comprobar *online* para detectar posibles errores. Esta solución es explotada en la técnica de *Predicción del Contador de Programa* descrita en este capítulo en el apartado 3.2.

Una segunda solución consiste en optimizar la información necesaria para realizar la comprobación del flujo de instrucciones. En las soluciones existentes, dicha información consiste en un conjunto de firmas que se calculan generalmente en tiempo de compilación y se almacenan en el sistema. En el apartado 3.3 se propone una solución optimizada en la que sólo una porción de las firmas se almacena en el sistema, las cuales además se pueden calcular en tiempo de ejecución. De esta forma se reducen drásticamente las necesidades de memoria para almacenar todas las firmas y el esfuerzo necesario para gestionarlas. En particular, esta técnica proporciona una solución efectiva para el caso de aplicaciones de gran tamaño, que son habituales en los procesadores actuales.

Finalmente, una tercera alternativa es aumentar la observabilidad del sistema, de manera que se obtenga mayor información de la ejecución y no sea preciso almacenar información en tiempo de compilación. Para ello es necesario añadir un punto de observación adicional. Una posibilidad es utilizar la interfaz de depuración como punto de observación de la ejecución. Las infraestructuras de depuración son utilizadas para realizar

operaciones de depuración durante la fase de desarrollo y son ampliamente utilizadas en microprocesadores modernos. Debido a que estas unidades de depuración están inoperantes durante la ejecución normal de una aplicación, pueden ser fácilmente reutilizadas para observación *online* de la ejecución sin coste adicional. Por otro lado, permiten observar el funcionamiento del microprocesador sin interferir en su funcionamiento y sin la necesidad de realizar modificaciones hardware o software. Como consecuencia de todas estas características, algunos trabajos han propuesto la utilización de la interfaz de depuración como punto de observación para llevar a cabo la monitorización de la ejecución [Gros10],[Gall09]. Tanto en [Gros10] como en [Gall09] se presentan trabajos que implementan técnicas para detectar errores en microprocesador, anteriormente estudiadas, aportando como innovación la utilización de la interfaz de depuración para obtener los datos necesarios para la implementación de dichas técnicas. Los resultados que se presentan en [Gros10], [Gall09] muestran la validez de la interfaz de depuración para llevar a cabo la observación del sistema. Las técnicas que se proponen en esta tesis aprovechan las ventajas de la interfaz de depuración, más concretamente, de la *interfaz de traza,* para obtener la información necesaria acerca del flujo de instrucciones en ejecución.

Las técnicas desarrolladas a lo largo de la tesis han sido implementadas principalmente en un sistema basado en el microprocesador *LEON3* [Gais01], como ejemplo paradigmático de microprocesador que dispone de una interfaz de traza. Este microprocesador, al igual que la mayoría de los microprocesadores modernos, cuenta con una infraestructura de depuración y en particular con una interfaz de traza que proporciona información acerca de cada instrucción ejecutada en tiempo real. La interfaz de traza se conecta habitualmente a una memoria o *buffer* circular, denominado *buffer de traza*, que almacena la información más relevante de la ejecución, como por ejemplo, el contador de programa y el código de operación. El *buffer de traza* es utilizado durante la fase de depuración para conocer las últimas instrucciones ejecutadas por el microprocesador en cualquier momento, lo que resulta especialmente útil para la depuración

de aplicaciones complejas. La monitorización puede hacerse a partir de la información almacenada en el *buffer de traza,* o desde la interfaz de traza directamente, siendo esta última opción la preferida dado que la observación es más efectiva.

Todas las técnicas descritas a lo largo de este capítulo obtienen la información de la ejecución realizando la observación a través de la *interfaz de traza*. De esta manera se puede realizar una monitorización online de la ejecución de una manera eficiente, no intrusiva y a muy bajo coste, ya que se está reutilizando un hardware ya existente. Si además se combina con el punto de observación habitual, que es la interfaz de la memoria de instrucciones, se puede aumentar la observabilidad del sistema de manera que no es necesario almacenar información adicional. Basándose en esta idea, en el apartado 3.4 se propone la técnica de *Monitorización Dual*, la cual permite detectar todos los errores de control de flujo de una manera muy eficiente.

Para la implementación de cada una de las técnicas propuestas se ha utilizado la arquitectura genérica que se muestra en la Figura 1. Esta arquitectura consta básicamente de tres partes: un bloque encargado de la interfaz con el microprocesador, un segundo conjunto de bloques que implementan cada una de las técnicas propuestas, y un tercer bloque encargado de coordinar y controlar el correcto funcionamiento de los diferentes elementos que componen la arquitectura. El bloque *Interfaz* es el bloque encargado de capturar la información de interés sobre las instrucciones ejecutadas, en concreto de la interfaz de traza, decodificar dicha información y proporcionarla al resto de bloques. El bloque Control se encarga de coordinar el funcionamiento de los distintos bloques, gestionar posibles estados de error, coordinar accesos a memoria y gestionar la máquina de estados que controla el funcionamiento del módulo. Por último, la funcionalidad específica de cada una de las técnicas que se han desarrollado ha dado lugar a diferentes bloques con características particulares. Cada uno de estos bloques es detallado en los distintos sub-

apartados de este capítulo correspondientes a las diferentes técnicas desarrolladas en la tesis doctoral.

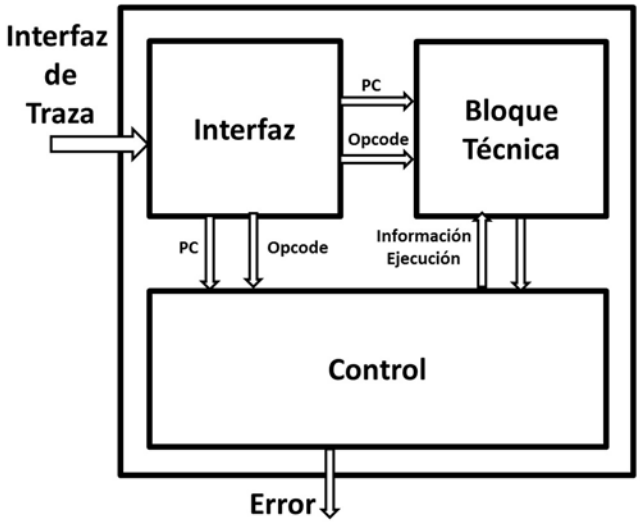

Figura 8: Arquitectura general del módulo hardware externo que implementa las técnicas de CFC propuestas.

A continuación se describen en detalle cada una de las técnicas que se han desarrollado a lo largo de la tesis doctoral y que son aportación original de este trabajo: *Predicción del Contador de Programa*, *Monitorización de Firmas y Monitorización Dual*. Se realiza una descripción teórica de cada una de las técnicas, destacando las aportaciones más importantes de cada una de ellas, así como sus principales ventajas e inconvenientes. Los resultados experimentales obtenidos con estas técnicas se presentarán en los capítulos posteriores.

3.2. Predicción del Contador de Programa

La técnica de *Predicción del Contador de Programa* [Parr11], [Parr13], [Parr14a], [Parr14b], [Boya15] (en inglés *PC Prediction)*, es una técnica de *CFC* que consiste en predecir el valor siguiente del contador de programa analizando el código de operación de la instrucción ejecutada y teniendo en cuenta el valor actual del contador de programa. El valor predicho se compara con el valor actual del contador de programa, de manera que se detecta un error si ambos valores no coinciden.

Para implementar esta técnica es necesario disponer de un interfaz de observación del flujo de ejecución que proporcione al menos los valores del contador de programa y del código de operación para cada instrucción. Existen dos posibles interfaces para este propósito: la interfaz de acceso a la memoria de programa o cache de instrucciones, y la *interfaz de traza*. El primero permite obtener el contador de programa en la fase de búsqueda de la instrucción, antes de que la instrucción se ejecute, mientras que el segundo permite obtener el contador de programa cuando la instrucción ya se ha ejecutado. Aunque en principio ambas interfaces son factibles, la segunda es preferible ya que filtra posibles efectos que pueden ocurrir durante la ejecución, tales como anulación de instrucciones o errores que pueden ocurrir durante la ejecución misma en cualquiera de las etapas del *pipeline*.

3.2.1. Conceptos básicos

Para poder entender la técnica de *Predicción del contador de programa* es preciso introducir algunos conceptos básicos sobre los elementos que forman parte típicamente de la arquitectura de un microprocesador, como la arquitectura de ejecución en *pipeline*, el contador de programa y el código de operación.

El contador de programa es un registro que forma parte del núcleo de un procesador y que indica la posición en la que se encuentra el procesador en la ejecución de la secuencia de instrucciones. El contador de programa se incrementa de manera automática en cada ciclo de instrucción, de manera que las instrucciones se ejecutan secuencialmente. Esta ejecución secuencial se ve alterada por las instrucciones que afectan al flujo de control, tales como saltos, bifurcaciones y llamadas a subrutinas. Para poder calcular el contador de programa correctamente es necesario conocer el código de operación de la instrucción que se esté ejecutando. El código de operación es la parte de una instrucción que especifica la operación que debe ser ejecutada. Su formato está determinado por la arquitectura del conjunto de instrucciones del procesador. Una instrucción de lenguaje máquina contiene un código de instrucción y opcionalmente uno o varios

operandos, sobre los que la instrucción tiene que llevar a cabo la operación. Estos operandos pueden ser, dependiendo de la arquitectura, registros, direcciones de memoria, valores almacenados en la pila, etc… Un código de operación puede especificar operaciones aritméticas, lógicas, de transferencia de datos y operaciones que produzcan un cambio sobre el flujo de programa, y que por tanto, lleven un control sobre la ejecución del programa.

Para poder ejecutar una instrucción se tienen que llevar a cabo una serie de operaciones, como la propia búsqueda de la instrucción en memoria, la decodificación, etc… que pueden ser ejecutadas de manera secuencial, de manera que cada etapa es ejecutada en un único ciclo de reloj. Se requeriría, por tanto, un número de ciclos de reloj igual al número de etapas necesarias para ejecutar la instrucción. La arquitectura de ejecución en *pipeline* consiste en aprovechar esta posibilidad de ejecución secuencial, de manera que se puedan estar procesando distintas instrucciones simultáneamente y cada una de ellas en una etapa diferente. En cada ciclo de reloj se están procesando un número de instrucciones igual al número de etapas de *pipeline* y cada una en una etapa distinta. Se consigue así optimizar la ejecución de manera que en cada ciclo de reloj se finaliza la ejecución de una instrucción en lugar de necesitar un número de ciclos de reloj igual al número de etapas para llevar a cabo la ejecución de cada instrucción.

Como se ha comentado anteriormente, cada una de las técnicas desarrolladas a lo largo de la tesis doctoral han sido implementadas en el microprocesador *LEON3,* el cual cuenta con una arquitectura de *pipeline* de 7 etapas. A continuación se detallan cada una de las etapas para este procesador:

- FE (Instruction Fetch): etapa en la que se realiza la búsqueda de la instrucción en memoria. La instrucción se obtiene de la memoria caché, si está disponible, o de la memoria principal. Al final de esta etapa se obtiene una instrucción válida.

- DE (Decode): etapa en la que se realiza la decodificación la instrucción.
- RA (Register access): Se lleva a cabo el acceso al banco de registros con la finalidad de obtener los operandos sobre los que se va a llevar a cabo la operación.
- EX (Execute): etapa en la que se realizan las distintas operaciones, ya sean lógicas o aritméticas mediante la utilización de la *ALU* (*Unidad Aritmético-Lógica*). En el caso de instrucciones de acceso a memoria, se calcula la dirección.
- ME (Memory): etapa en la que se accede a la memoria de datos. Los datos obtenidos durante la etapa de ejecución se almacenan en la caché de datos o en la memoria principal.
- XC (Exception): etapa en la que se resuelven las interrupciones.
- WR (Write): en esta etapa el resultado de las operaciones que se han llevado a cabo, ya sean lógicas, aritméticas u operaciones de lectura de memoria, se almacenan en el banco de registros.

3.2.2. Método para el cálculo del contador de programa

La técnica de *Predicción del contador de programa* predice la dirección de memoria de la próxima instrucción a ejecutar, o lo que es lo mismo, el valor siguiente del contador de programa. Para poder realizar dicha predicción es necesario decodificar la instrucción con la finalidad de determinar si la ejecución de dicha instrucción afecta al flujo de programa o no. Una vez decodificada la instrucción, teniendo en cuenta la manera en la que la instrucción afecta al flujo de programa y conocido el valor actual del contador de programa se procede a realizar la predicción del próximo valor. Es importante destacar que no es necesario realizar una decodificación completa de las instrucciones, sino tan sólo identificar aquellas instrucciones que afectan al control del flujo, lo que es mucho más simple.

La metodología seguida para predecir el valor del siguiente contador de programa se ilustra en la Figura 9. Una vez decodificada la instrucción se comprueba si la ejecución de dicha instrucción afecta al flujo secuencial de

la ejecución del programa. En el caso de instrucciones que no afectan al flujo de programa, el contador de programa debe de incrementarse con el tamaño de la instrucción. En caso contrario se debe de analizar la manera en la que la instrucción afecta al flujo de ejecución. Más adelante en este capítulo se realiza un análisis exhaustivo de las diferentes instrucciones que se pueden dar en el caso particular del microprocesador *LEON3*. En principio, se pueden simplificar las diferentes tipos de instrucción de salto en dos casos simplificados: instrucciones que producen un salto en la ejecución e instrucciones que no realizan el salto porque no se cumplen las condiciones para que se produzca el salto. En el primer caso, el contador de programa debe de incrementarse con el *offset* de salto, mientras que en el segundo caso el contador de programa es incrementado con el tamaño de la instrucción.

Una vez calculado el valor previsto del contador de programa, éste se compara con la dirección obtenida en tiempo de ejecución activando una señal de error en caso de que no sean iguales.

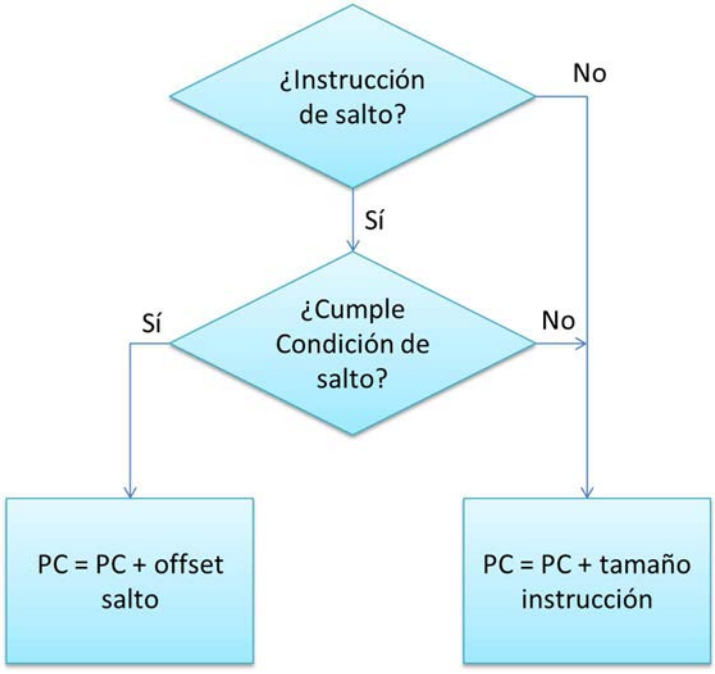

Figura 9: Esquema para calcular el contador de programa.

Para implementar la técnica *Predicción del contador de programa* en un sistema basado en el microprocesador *LEON3* ha sido necesario analizar

en detalle el conjunto de instrucciones de la arquitectura *SPARC* (*Scalable Processor ARChitecture*) [SPAR91]. La arquitectura *SPARC* define un conjunto de instrucciones reducido, y está basada en los diseños *RISC I* y *RISC II* (*Reduced Instruction Set Computer*). La arquitectura *SPARC* hace distinción entre instrucciones que no afectan al flujo de control, (*Integer Arithmetic Instruction, Load/Store instruction, Read/Write State Registers, Floating-Point Operate Instructions)*, e instrucciones que modifican o pueden llegar a modificar el flujo de programa, denominadas *Control-Transfer Instructions.* A continuación se detallan los diferentes tipos de *Control-Transfer Instructions* que implementa la arquitectura *SPARC*:

- Saltos condicionales (denominados *Conditional branch* en la arquitectura SPARC): las instrucciones de salto condicional calculan la dirección de destino como una dirección relativa al contador de programa. La dirección relativa es proporcionada como un operando de la instrucción. La condición del salto forma parte del código de instrucción y hace referencia a los bits del registro de estado. Las instrucciones de salto incondicional se consideran un caso particular de salto condicional, en el que la condición de salto es siempre cierta.
- Llamadas a subrutinas (denominadas *Call and Link o CALL)*: al igual que las instrucciones de salto condicional, la dirección del salto es calculada de manera relativa al contador de programa. En este caso no se tiene que cumplir cierta condición para que se lleve a cabo el salto, sino que éste se realiza siempre.
- *Jump and Link (JMPL)*: la instrucción de salto *Jump* realiza un salto a una dirección absoluta, determinada mediante direccionamiento indirecto.
- Interrupción o *Trap*: la instrucción de interrupción evalúa una condición proporcionada como un operando de la misma, y en el caso de que dicha condición se cumpla se genera un evento de interrupción. La instrucción a la que tiene que saltar es obtenida de una tabla, denominada *Trap Base Register* (*TBR*). La instrucción del contador de programa actual, y del siguiente contador de

programa son almacenadas, con el objetivo de que la ejecución continúe al regresar de la subrutina dedicada a la interrupción.

- *Return from trap (RETT)*: la instrucción de retorno de interrupción vuelve a cargar el valor del contador de programa que había antes de que se produjera el evento de interrupción, de manera que se continúa la ejecución por el mismo punto en el que se encontraba antes de la interrupción.

Como se puede observar en los diferentes tipos de instrucción descritos para la arquitectura *SPARC,* llevar a cabo la predicción del salto con absoluta certeza puede no ser factible en la práctica. Dependiendo del conjunto de instrucciones se tiene más o menos información directa acerca de la instrucción. En las instrucciones que utilizan condiciones de salto dinámicas o en las que la dirección de salto se obtiene de manera indirecta, una predicción certera del salto requeriría información que no está accesible desde la interfaz externa. De todas formas, aún en estos casos es posible hacer una predicción muy aproximada. Como regla general se pueden diferenciar dos casos principalmente, que son descritos a continuación:

- Condición de salto desconocida en salto condicional: se tienen dos posibles opciones del valor del contador de programa que se darán como válidas. Una de ellas es el contador de programa correspondiente al salto y el otro valor es el resultante de sumar al contador de programa anterior el tamaño de la instrucción. Se considera error si el valor del contador de programa que se obtiene a través de la *interfaz de traza* no se corresponde con ninguna estas dos alternativas. En esta situación, aquellos errores que producen un cambio en el registro de estado, dónde se almacenan los bits de condición, no se podrían detectar. En puridad, este tipo de errores puede ser considerado más bien como un error de datos que afecta al flujo de programa y puede ser detectado combinando la técnica de *Predicción del Contador de Programa* con una técnica de endurecimiento software.

- Dirección absoluta o relativa de salto desconocida en salto incondicional: la dirección de salto está almacenada en un registro que no es accesible desde la interfaz externa. En este caso se sabe que se tiene que llevar a cabo un salto, pero se desconoce la dirección de destino. Se debe comprobar que se realiza un salto y, por tanto, se dará por bueno cualquier valor del contador de programa que sea distinto del valor que tendría si no se realizara ningún salto. Este caso es más delicado, ya que cualquier error que afecte a los registros que almacenan la dirección de salto no será detectado, dando como válido el salto. De nuevo, este tipo de error corresponde más bien a la categoría de error de datos.

3.2.3. Implementación de la técnica de Predicción del contador de programa

Con la finalidad de probar la capacidad de detección de errores de la técnica de *Predicción del Contador de Programa* se ha desarrollado un módulo hardware que implementa dicha técnica. La Figura 10 muestra la estructura interna del módulo hardware.

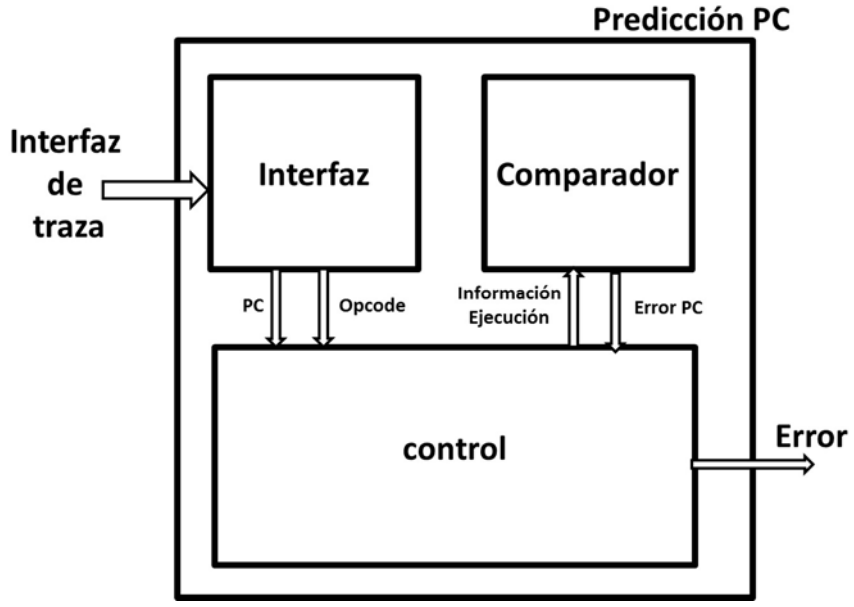

Figura 10: Módulo Hardware que implementa la técnica Predicción del contador de programa.

El módulo hardware ha sido subdividido en tres bloques principalmente: *Interfaz, Control y Comparador.* A continuación se describe brevemente la funcionalidad de cada uno de los bloques.

- *Interfaz*: Este bloque se encarga de obtener el contador de programa y el código de instrucción por medio de la interfaz de traza. Además de transferir al *Control* esta información, la interfaz se encarga de decodificar el código de operación y enviar la información decodificada al *Control*. Por tanto, determina si la instrucción es de salto o no, si es condicional o incondicional y calcula el *offset* de salto.
- *Control*: Máquina de estados encargada del correcto funcionamiento del resto de bloques. Se encarga de controlar si el módulo se encuentra en estado ocioso esperando el inicio de la parte crítica del programa a monitorizar, en funcionamiento y por tanto monitorizando la ejecución del programa, o en estado de error tras haberse producido un cambio inesperado en el flujo de control. A continuación se detallan las principales funciones de este bloque:
  - Indicar al bloque *Comparador* cuando se dispone de una nueva instrucción.
  - Proporcionar información al bloque *Comparador* acerca de la nueva instrucción ejecutada: si es una instrucción de salto, condicional o incondicional, el *offset* del salto, etc…
  - Coordinar el funcionamiento de los distintos bloques que componen el módulo.
  - Activar una señal de error en caso de que se haya detectado una variación en el contador de programa.
- *Comparador*: Bloque encargado de calcular el contador de programa a partir de la información decodificada obtenida del código de instrucción y compararlo con el contador de programa. Tanto la información decodificada del código de instrucción como el contador de programa son proporcionados por el *Control*. En caso de que ambos contadores de programa no fueran iguales

avisaría al bloque de *Control* a través de una señal de error. Este bloque sigue el esquema mostrado en la Figura 9 para el cálculo del contador de programa y tiene en cuenta el conjunto de instrucciones de la arquitectura *SPARC* descrito anteriormente.

3.2.4. Análisis de la técnica de Predicción del Contador de Programa

Uno de los principales puntos a destacar de la técnica *de Predicción del contador de programa* es que la observación de la ejecución se realiza a través de la *interfaz de traza*, es decir, que los valores del contador de programa y código de operación de la instrucción son obtenidos a través de dicha interfaz. Por tanto, los valores de ambas señales son proporcionados en la etapa de ejecución, permitiendo la detección de errores de flujo en cualquiera de las etapas anteriores que acaben produciendo una variación en el contador de programa.

En segundo lugar, cabe destacar que esta técnica no requiere almacenar información calculada en tiempo de compilación para llevar a cabo un control del flujo. En consecuencia, la cantidad de recursos que se requieren para su implementación son reducidos en comparación con otras técnicas de *CFC* que precisan de información online [OhSh02], [Berg09], [Yung05]. Esto es debido a que en ningún momento es necesario comparar la ejecución con una ejecución previa para saber si esta es correcta, sino que se aprovecha información que está implicita en cualquier secuencia de instrucciones que se ejecute. Si además tenemos en cuenta que la observación de la ejecución se lleva a cabo reutilizando las infraestructuras de depuración ya existentes, y por tanto con un coste muy bajo, se puede afirmar que la técnica de *Predicción del contador de programa* puede ser implementada con un incremento de área que puede llegar a ser despreciable en sistemas basados en microprocesador de aplicación real, como es el microprocesador *LEON3*.

Como se ha comentado anteriormente, está técnica es capaz de detectar un subconjunto de los errores que se producen en un

microprocesador, siendo altamente efectiva en la detección de aquellos errores que acaben produciendo una variación en el contador de programa. La capacidad de detección de errores depende de manera sustancial de la completitud de la información que se pueda obtener desde la interfaz de observación. En particular, cuando las direcciones o las condiciones de salto no están disponibles, no se puede predecir con precisión si un salto es correcto. En estos casos habría un pequeño número de errores que no podrían ser detectados por la técnica propuesta. Sin embargo, podrían ser fácilmente detectados si se trabaja en conjunto con otras técnicas de *CFC* y/o técnicas de endurecimiento software. Los resultados experimentales, mostrados en los capítulos 4 y 5, muestran que la técnica de *Predicción del contador de programa* alcanza muy buenos resultados cuando es implementada en combinación con la técnica *Monitorización dual,* descrita más adelante en este capítulo, alcanzando una detección del 100% de los errores que afectan al contador de programa y al código de operación.

3.3. Monitorización de firmas

La técnica de *Monitorización de firmas* [Parra11], [Boya13], [Boya14], [Boya15] consiste en el cálculo y posterior comprobación de una firma o resultado compactado de la ejecución de un programa o sección de código. Esta técnica pertenece a un conjunto de técnicas ampliamente estudiadas por la comunidad científica conocidas generalmente como *Signature monitoring* [Sonz11], [Alkh99], [OhSh02], [Vemu06], [Mich91], [Berg09], [Wilk90], [Yung05]. La solución propuesta en esta tesis pretende ampliar y mejorar esta técnicas de monitorización de firmas basándose en dos aspectos principales: el uso de la interfaz de traza, como medio de observación del flujo de instrucciones, y la optimización de los recursos hardware necesarios en relación a la capacidad de detección de errores, buscando un compromiso eficiente entre ambos.

En las técnicas basadas en *Signature monitoring,* el programa a ejecutar se divide en *bloques básicos*. Cada bloque tiene asignado una firma que se calcula en tiempo de compilación y que se almacena en el sistema.

Dicha firma es un valor pseudoaleatorio que se calcula compactando los códigos de instrucción de cada una de las instrucciones ejecutadas dentro de un mismo bloque básico. Para ello se utiliza generalmente un MISR (*Multiple Input Signature Register*), el cual se puede implementar mediante hardware o software. Al final de la ejecución de cada bloque básico se compara la firma calculada en tiempo de ejecución con la firma almacenada. Cualquier error que produjera una variación en los códigos de instrucción o en el orden de las instrucciones produciría una firma diferente, y por tanto sería detectado.

Como se ha mencionado anteriormente, una de las principales diferencias de la técnica propuesta respecto a otras técnicas existentes está en el punto de observación. La técnica de *Monitorización de firmas* propuesta en esta tesis realiza la observación de la ejecución a través de la *interfaz de traza*, de manera que los valores del contador de programa y del código de instrucción corresponden a la etapa de ejecución. Esto permite detectar errores de control de flujo que se hayan producido en cualquiera de las etapas del *pipeline*. Adicionalmente, se ha desarrollado además una mejora de la técnica, denominada *Address Checking*, que será descrita a lo largo de este capítulo.

La técnica de *Monitorización de firmas* ha sido desarrollada en el marco de una investigación conjunta entre la Universidad Carlos III de Madrid y el Politecnico di Torino (Italia).

3.3.1. Bloques Básicos

El primer paso para implementar la técnica de *Monitorización de firmas* es dividir el programa en bloques básicos [Alkh99], [Golo03], [OhSh02], [Venk03]. Un bloque básico es un conjunto de instrucciones libre de saltos intermedios. Todas las instrucciones que pertenecen a un bloque básico son siempre ejecutadas de manera secuencial una vez que se alcanza la primera instrucción del bloque. La única instrucción que puede ser de salto o que varíe el flujo de programa es la última, mientras que la única instrucción que puede ser destino de un salto es la primera.

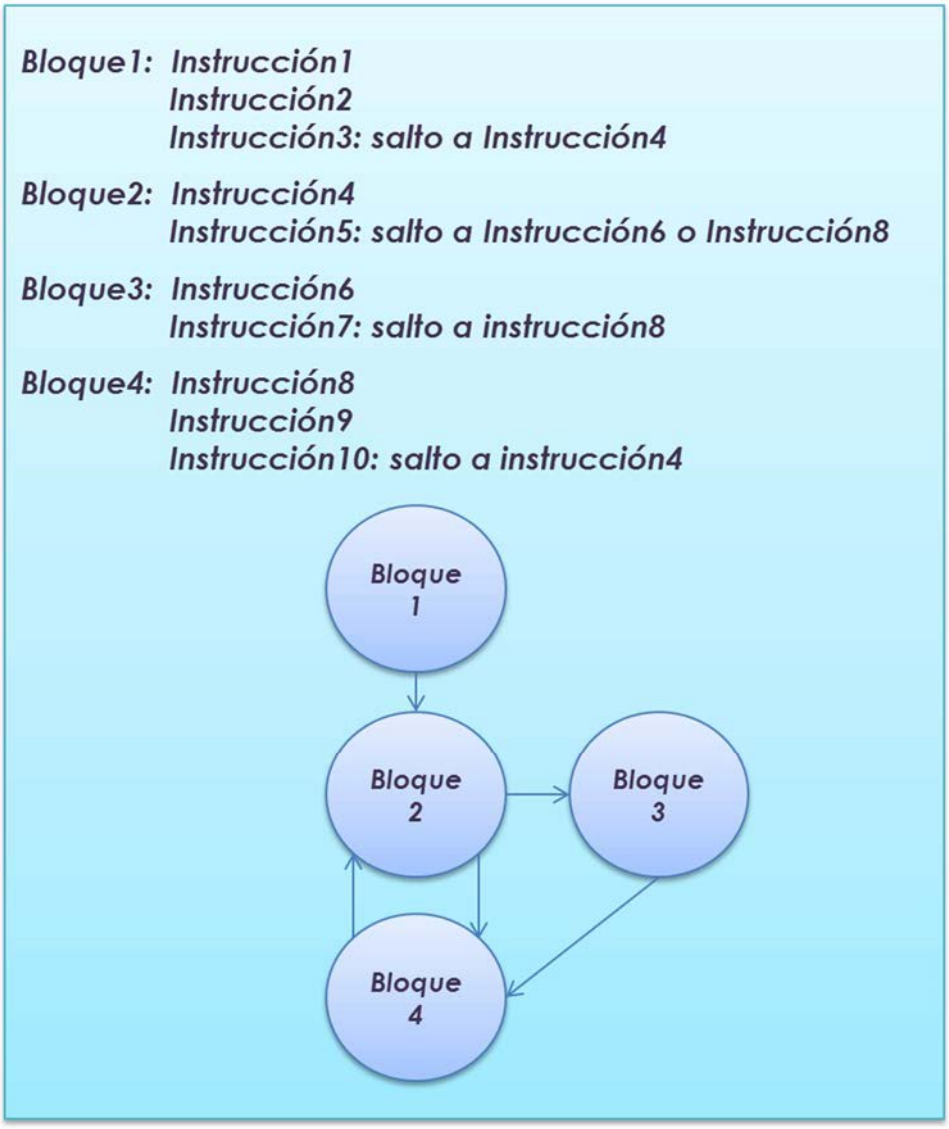

Figura 11: División en bloques básicos de un programa.

La Figura 11 muestra un ejemplo básico de la metodología seguida para dividir un programa en bloques básicos. Cada bloque está formado por un conjunto de instrucciones que son ejecutadas de manera secuencial. Esta característica se puede observar en el bloque 1, donde el programa ejecuta las instrucciones de la 1 a la 3 sin saltos intermedios. Únicamente la última instrucción puede ser una instrucción de salto. En el caso del bloque 1, la instrucción 3 es un salto incondicional a la instrucción 4. También se puede observar como la única instrucción de un bloque que puede ser el destino de un salto es la primera instrucción del mismo. En este ejemplo, las primeras

instrucciones de los bloques 2, 3 y 4 son los destinos de los saltos procedentes de otros bloques.

Un código se puede dividir en bloques básicos de manera dinámica o estática. Para dividir un código por bloques de manera dinámica se debe realizar una ejecución del mismo e ir comprobando donde se producen los saltos. Este método es incompleto puesto que no tiene en cuenta los saltos que no son tomados, y por tanto el número de bloques resultante puede ser menor. Para un análisis exhaustivo del número de bloques básicos en los que se divide un código se debe realizar un análisis estático. En este caso se realiza un análisis de cada una de las instrucciones del código a partir del código fuente en ensamblador, identificando cada uno de los posibles saltos o instrucciones que podrían afectar al flujo de programa. Se deben determinar también las direcciones de salto para poder identificar los inicios de los bloques básicos.

3.3.2. Cálculo de la firma

Cada bloque tiene una firma asignada y calculada en tiempo de compilación, a la que se le denomina firma de referencia. La firma se calcula compactando los códigos de instrucción ejecutados por el microprocesador. Al final de la ejecución de cada bloque se compara la firma calculada con la firma de referencia. Si estas firmas difieren, se debe activar una señal que indica que se ha producido una variación en el flujo de programa durante la ejecución y por tanto un error.

La firma se genera típicamente mediante MISR "*Multiple Input Signature Register*", que es un tipo de *LFSR* "*Linear Feedback Shift Register*". Un *LFSR* es un registro de desplazamiento cuya entrada proviene de aplicar al estado anterior una transformación lineal. Se parte de un valor semilla o valor inicial y la secuencia de valores que se generan depende completamente del estado anterior. En el caso de un MISR, la secuencia de valores depende del estado anterior y de la entrada disponible en cada momento. El resultado es una firma o valor compactado que tiene la propiedad que es distinta si alguno de los datos de entrada utilizados para generarla varía. De

esta manera, si se produce un fallo en el flujo de programa, los códigos de instrucción utilizados para generar la firma serán distintos y la firma será diferente a la calculada en tiempo de compilación. Gracias a esta propiedad, se puede almacenar toda la información relativa al flujo de un bloque básico en una única firma y comparando dicha firma se puede determinar si ha habido una variación del flujo de programa, sin necesidad de comparar instrucción a instrucción.

Para que un error dentro de un bloque básico pueda ser detectado es preciso que en algún momento de la ejecución se alcance el final del bloque, para que de esta manera se proceda a comparar la firma online con la firma de referencia. Si a consecuencia del error no se alcanzara nunca el final de un bloque básico, no se llegaría a realizar la comparación y el error no podría ser detectado. Un claro ejemplo en el que se produce este comportamiento es cuando se pierde de secuencia en la ejecución como consecuencia del error. Para detectar este tipo de errores se ha propuesto una mejora de la técnica denominada *Address Checking,* descrita más adelante en este capítulo.

3.3.3. Implementación de la técnica *Monitorización de firmas*

Se ha desarrollado un módulo hardware externo que implementa la técnica de *Monitorización de firmas*. En la Figura 12 se muestra la estructura interna de dicho módulo.

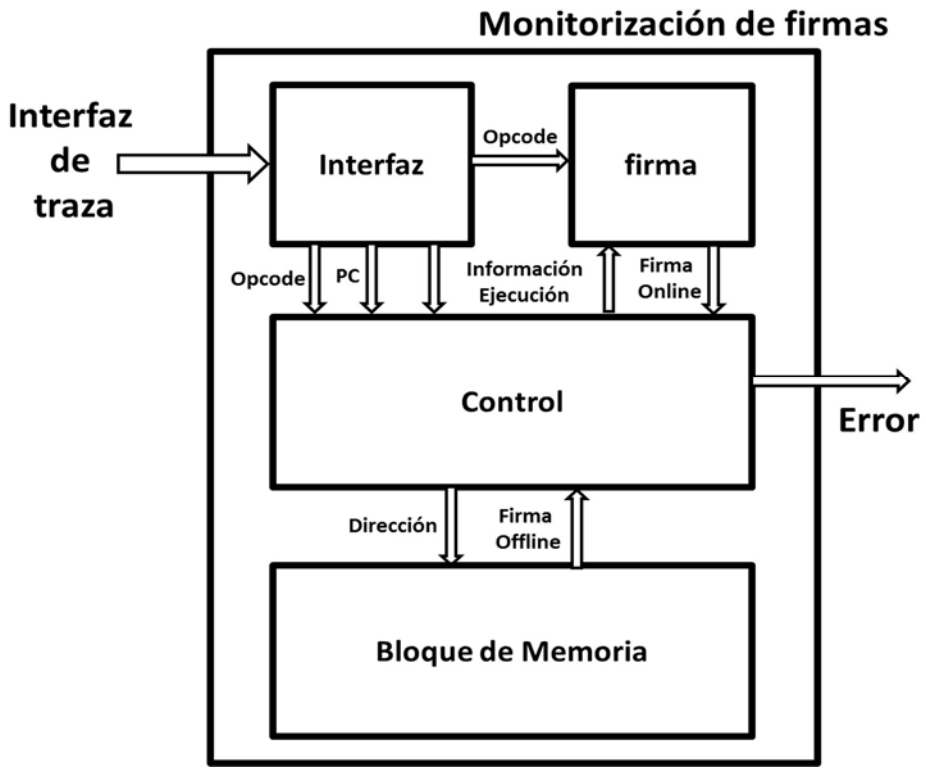

Figura 12: Módulo Hardware que implementa la técnica de Monitorización de firmas.

A continuación se describe brevemente la funcionalidad de cada uno de los bloques que forman parte del módulo:

- *Interfaz*: Bloque encargado de captar a través de la *interfaz de traza* la información más relevante de la ejecución del microprocesador y transferir dicha información al bloque de *Control*. Además proporciona al bloque firma el código de instrucción para que pueda calcular la firma on-line.
- *Firma*: Bloque encargado de calcular la firma on-line y proporcionar dicha firma al bloque de *Control* al finalizar la ejecución de un bloque básico. El bloque firma capta el código de instrucción cada vez que el control le indica que se ha ejecutado una nueva instrucción, y lo utiliza para generar una firma pseudoaleatoria.
- *Control*: Máquina de estados encargada del correcto funcionamiento del resto de bloques. Las principales funciones del control son las siguientes:
    - Indicar cuando finaliza e inicia un nuevo bloque.

  - Comparar la firma on-line con la firma de referencia, almacenada en el bloque de memoria, al finalizar la ejecución de un bloque básico.
  - Proporcionar información de la ejecución al bloque *Firma*, como por ejemplo, indicar cuando se produce una nueva instrucción o cuando debe de iniciarse el cálculo de una nueva firma.
  - Se encarga de gestionar el acceso al *Bloque de memoria* para obtener la firma de referencia del bloque correspondiente.
  - Coordina el funcionamiento de los distintos bloques del módulo.
  - Indica si ha habido un error en el flujo de programa a través de una señal de error.
- *Bloque de Memoria*: Bloque que almacena las firmas de referencia, que posteriormente serán utilizadas durante la ejecución para compararlas con las firmas calculadas en tiempo de ejecución.

3.3.4. Mejora de la técnica: Address Checking

Como se ha comentado anteriormente, una de las posibles limitaciones que presenta la técnica *Monitorización de firmas* es que es necesario alcanzar el final del bloque básico para que la comparación de firmas tenga lugar y que, por tanto, un error durante la ejecución de dicho bloque pueda ser detectado. Un error típico que produce este efecto es por ejemplo, la pérdida de secuencia de la ejecución. Una pérdida de secuencia comúnmente se produce cuando tiene lugar un salto a una dirección inexistente de la memoria de programa, de manera que la ejecución se bloquea y nunca se alcanza el final del bloque básico, ni se inicia la ejecución de uno nuevo.

Existen diferentes soluciones para detectar este tipo de problemas. Una de las posibles soluciones, muy utilizada en la literatura, es implementar un *watchdog timer* [Ashr07]. Esta solución consiste principalmente en llevar la cuenta de ciclos de reloj que transcurren entre la ejecución de dos instrucciones consecutivas, estableciendo un umbral a partir del cual se

considera que ha habido un error en la ejecución. De esta manera, cuando se produzca una pérdida de secuencia el contador se acabará desbordando y se producirá la detección de dicho error. La alternativa propuesta en este trabajo de tesis y a la que hemos denominado *Address checking*, consiste en comprobar que no se produzcan saltos ilegales dentro de un mismo bloque básico. Para ello es necesario capturar el valor del contador de programa al inicio de cada bloque y comprobar que cada vez que llegue una nueva instrucción el contador de programa se incrementa con un tamaño de offset igual al tamaño de la instrucción ejecutada, es decir, comprobar que no se ha producido ningún salto. El contador de programa calculado se compara con el contador de programa obtenido a través de la *interfaz de traza* con el fin de detectar errores en el flujo de programa que hayan provocado un salto dentro del bloque básico.

Con esta mejora se consigue detectar un mayor número de errores, ya que se pueden detectar aquellos errores que producen una pérdida de secuencia durante la ejecución. En los casos en los que se pierde la secuencia es muy probable que nunca se alcance el final del bloque básico que se esté ejecutando en ese momento y, por tanto, nunca se llegaría a realizar la comparación de firmas y el error no se llegaría a detectar. Sin embargo, utilizando *Address Checking* sí se podrían detectar este tipo de errores, ya que normalmente la secuencia se ha perdido porque se ha realizado un salto a una dirección inválida. En caso de producirse dicho salto se detectaría como un salto ilegal dentro de un bloque básico. Por otro lado, utilizando esta mejora se consigue disminuir la latencia de detección de errores. La técnica de *Monitorización de firmas* requiere que se llegue al final del bloque básico para poder comparar las firmas y determinar si la ejecución es correcta. En el peor de los casos la latencia será igual al tamaño del bloque básico, cuando el error se produce en la primera instrucción del mismo. Utilizando *Address Checking* la latencia es mínima cuando se produce un salto ilegal, ya que se detecta al momento, sin necesidad de ejecutar el resto del bloque.

3.3.5. Metodología de almacenamiento y acceso a las firmas

Una de los principales inconvenientes de este método es la necesidad de almacenar la información calculada en tiempo de compilación, es decir, la necesidad de almacenar las firmas de referencia. Esto supone un impacto sobre el área del sistema que no es significativo en aplicaciones simples, pero que puede tener un impacto importante en aplicaciones reales. Es por tanto necesario buscar una metodología de almacenamiento y acceso a las firmas lo más eficiente posible. La opción más simple es utilizar una memoria lo suficientemente grande como para almacenar todo el rango de direcciones utilizadas en el programa y hacer un direccionamiento utilizando la dirección de inicio de bloque. Esta opción requiere una memoria de gran tamaño en aplicaciones reales.

La solución que se propone en esta tesis, y que es una aportación original, consiste en limitar el almacenamiento de información, de manera que sólo un subconjunto de las firmas está disponible en un momento dado. Evidentemente, esta solución implica que no siempre se podrá comprobar si una firma es correcta. Sin embargo, tomando como inspiración la arquitectura usada en las memorias cachés asociativas, se puede conseguir una elevada tasa de aciertos con un almacenamiento de información reducido.

Para implementar esta solución, se proponen dos posibles alternativas. La primera de ellas es una solución estática basada en la implementación de una tabla (llamada *CFC Signature Table*, o *CFC-ST*) compuesta de un conjunto de entradas fijo. Cada entrada almacena la información de un bloque básico y es pre-calculada en tiempo de compilación. En particular, cada entrada almacena la siguiente información:

- La dirección de la primera instrucción del bloque básico.
- La firma calculada en tiempo de compilación correspondiente al bloque básico.

Cada entrada de la *CFC-ST* tiene una correspondencia con los bits menos significativos de la dirección de inicio del bloque básico. Utilizando

esta correspondencia se consigue realizar un acceso a las diferentes entradas de la *CFC-ST* de una manera más rápida y eficiente. Además, se puede implementar la CFC-ST con una memoria de pequeño tamaño. Este método de acceso requiere analizar el código de la aplicación antes de su ejecución y realizar las siguientes tareas:

- Identificar cada uno de los bloques básicos en los que se divide el código.
- Calcular las direcciones de inicio y las firmas para cada bloque básico.
- Almacenar las direcciones de inicio y las firmas en las entradas adecuadas de la *CFC-ST*.

Durante la ejecución de un programa, cada vez que el procesador alcanza el final de un bloque básico se accede a la entrada correspondiente y se realiza la comparación de firmas. Sin embargo, debido a que el tamaño de la *CFC-ST* es limitado, dos o más bloques básicos pueden tener los mismos n bits menos significativos, siendo por tanto asociados a la misma entrada de la *CFC-ST* y provocando una colisión. En este caso, en cada acceso a la *CFC-ST* se comprueba si está almacenada la información del bloque básico correcto, utilizando la dirección de la primera instrucción del bloque. Si la información correspondiente al bloque no está disponible, se da el acceso como erróneo y no se realiza la comprobación.

Para aplicaciones de pequeño tamaño, en términos del número de bloques básicos, se puede almacenar una tabla completa. En caso contrario, se pueden utilizar varias técnicas para priorizar los bloques básicos que se almacenarían en la tabla, de acuerdo a diferentes criterios:

- Por tamaño del bloque básico: se eligen los bloques de mayor tamaño, dado que la probabilidad de que un fallo afecte a un bloque básico es proporcional a su tamaño.
- Por frecuencia de ejecución: se eligen los bloques que se ejecutan más frecuentemente. La frecuencia se estima mediante ejecuciones de prueba con casos típicos.

- Por tamaño y frecuencia de ejecución: se eligen los bloques que tienen un mayor valor del producto de los dos parámetros anteriores.

Para mejorar el método de acceso se utiliza una arquitectura similar a la utilizada en una caché asociativa. Se organizan las entradas de la *CFC-ST* en conjuntos de *m* entradas, donde *m* suele tener un valor de 2 o 4. La información de un bloque básico puede ser almacenada en cualquier posición del conjunto que le corresponda. Cuando se alcanza el final de un bloque básico durante una ejecución, se accede al conjunto que corresponda de la *CFC-ST* y se comprueba si alguna de la información almacenada se corresponde con el bloque que está siendo ejecutado. Esta solución incrementa ligeramente la complejidad de la *CFC-ST,* pero proporciona una mejora en la cobertura de fallos.

La segunda solución que se propone implementa un método dinámico basado en una actualización dinámica de la tabla. En lugar de calcular las firmas en tiempo de compilación, éstas se calculan durante la ejecución del programa. Inicialmente la CFC-ST se encuentra vacía. Cada vez que se ejecuta un bloque, si la firma no se encuentra en la tabla se almacena la firma calculada junto con la dirección de inicio del bloque básico. Para resolver las colisiones se utiliza un algoritmo de reemplazo similar al algoritmo *LRU* (*Least Recently Used*) utilizado en memorias caché. El algoritmo de reemplazo *LRU* descarta el elemento que es menos usado recientemente, por tanto, aplicándolo a la tabla *CFC-ST* se descartaría la entrada de la tabla que ha sido usada en menor medida recientemente en la simulación. Para poder aplicar correctamente este algoritmo es preciso llevar un control de qué elementos se han usado y cuando han sido usados. La implementación general en memorias cachés utiliza una serie de bits que contabilizan la antigüedad de cada una de las entradas de la memoria. Estos bits deben de ser actualizados cada vez que se modifica una de las entradas de la memoria.

A diferencia del método estático, la tabla *CFC-ST* utilizada en el método dinámico no requiere un pre-análisis del software, sino que iniciaría la ejecución vacía y se iría rellenando según fuera avanzando la ejecución.

Respecto a la cobertura de fallos, el método dinámico no puede detectar un fallo en un bloque básico que se ejecute por primera vez o que no se haya ejecutado recientemente. Sin embargo, el método dinámico tiene la ventaja de que continuamente tiende a almacenar en la tabla *CFC-ST* la información de los bloques básicos que han sido utilizados más recientemente, de manera que se realiza una optimización del espacio disponible de una manera intrínseca. Gracias a este escenario, se consigue alcanzar una elevada cobertura de detección de fallos incluso cuando hay una diferencia importante entre el tamaño disponible en la tabla *CFC-ST* y el número de bloques básicos en los que se puede dividir el software. Este comportamiento se puede observar en los resultados experimentales que se han obtenido y que se muestran en el capítulo 4.

3.3.6. Análisis de la técnica de Monitorización de Firmas

La segunda técnica propuesta en esta tesis doctoral es la técnica de *Monitorización de firmas* y se basa principalmente en compactar la información de la ejecución de cada uno de los bloques básicos en una firma y compararla con una firma de referencia calculada en tiempo de compilación. Esta técnica ha sido desarrollada en colaboración con el Politecnico di Torino y constituye una de aportaciones originales de esta tesis doctoral.

La técnica *Monitorización de firmas* realiza la monitorización de la ejecución a través de la *interfaz de traza*, lo que posibilita la detección de errores que se produzcan en cualquiera de las etapas del pipeline y que acaben afectando al flujo de programa. Esta técnica es capaz de alcanzar un porcentaje de detección de errores que afectan al flujo de programa cercanos al 100% cuando trabaja en combinación con la técnica *Address Checking*, tal y como se muestra en los resultados experimentales descritos en el capítulo 4. Gracias a la técnica *Address Checking* se consiguen detectar errores que escapan a la comparación de firmas y que, por tanto, no pueden ser detectados por la técnica *Monitorización de firmas.* Además, la latencia de detección se ve ampliamente reducida ya que se comprueba para cada instrucción ejecutada dentro de un mismo bloque

básico que el contador de programa tenga el valor esperado. Gracias a esta comprobación cualquier salto ilegal dentro de un bloque básico es detectado con la mínima latencia posible.

Uno de los principales problemas que presentan las técnicas basadas en *Signature Monitoring* es la necesidad de realizar un estudio previo del software a ejecutar, así como la necesidad de almacenar la información obtenida en dicho estudio preliminar. Esto se traduce en un incremento del área necesaria para implementar este tipo de técnicas, que puede llegar a ser importante en aplicaciones complejas. Uno de los objetivos de esta tesis doctoral es intentar reducir este impacto sobre el rendimiento del sistema, de manera que se consiga alcanzar una cobertura a fallos importante reduciendo el impacto. Por esta razón, se han propuesto dos posibles alternativas a la hora de gestionar la información de referencia. La primera de ellas consiste en un método estático y la utilización una tabla de tamaño reducido. En este primer método se debe buscar una solución de compromiso entre el tamaño de la tabla y el número de errores que se quedan sin detectar como consecuencia de las colisiones que se producen. En cualquier caso se consigue alcanzar una tasa de detección de errores óptima reduciendo el impacto sobre el rendimiento del sistema de una manera importante. La segunda solución propuesta actualiza la tabla utilizando un método dinámico. Gracias al método propuesto no es necesario realizar un análisis previo del software a ejecutar y además se consigue reducir de manera significativa el tamaño de la tabla y, por tanto, el impacto sobre el rendimiento del sistema.

Basándonos en los resultados experimentales obtenidos en las diferentes campañas de inyección realizadas a lo largo de la tesis doctoral se puede concluir que la técnica *Monitorización de firmas* trabajando en conjunto con la técnica *Address Checking* consigue alcanzar una detección de errores importante, siendo cercana al 100% cuando los errores afectan únicamente al flujo de programa. Además, gracias a los dos métodos de acceso a memoria propuestos se consigue reducir significativamente el área

necesaria para la implementación de la técnica, reduciendo el impacto sobre el rendimiento del sistema.

3.4. Monitorización Dual

Básicamente, las técnicas de monitorización consisten en obtener información en tiempo real de la operación del procesador a través de un interfaz apropiado y compararla con la ejecución esperada para detectar errores en el flujo de control. Aun en el supuesto de que la interfaz proporcione información suficiente, en cualquier caso es necesario almacenar y acceder grandes cantidades de información acerca de la ejecución esperada.

La técnica de *Monitorización dual* [parra14b] se centra en aumentar la observabilidad para asegurar una detección de errores eficiente con el menor impacto posible sobre el rendimiento del sistema. Para alcanzar este objetivo se realiza la monitorización de la ejecución desde dos puntos de observación diferentes. El primer punto de observación se sitúa en el bus de la memoria [Bens03] o el bus de la memoria caché [Berg09]. Este es el punto de observación que se ha utilizado tradicionalmente Este punto de observación es el que se ha utilizado tradicionalmente en la mayoría de las técnicas que utiizan una monitorización externa. El segundo punto de observación es la *interfaz de traza*, propuesta recientemente en algunos trabajos [Gros10], [Gall09]. Cada uno de estos puntos de observación proporciona información acerca de la ejecución del programa en diferentes instantes de la misma. Por lo tanto, si se compara la información que se ha obtenido en ambos puntos de observación se pueden detectar cambios que se hayan podido producido en el flujo de instrucciones como consecuencia de un error.

La principal ventaja que se deriva de la utilización de dos puntos de observación diferentes es que no es necesario realizar un análisis previo del software con la finalidad de obtener información de la ejecución para realizar una posterior comparación. Toda la información necesaria para validar el flujo de programa se obtiene en tiempo de ejecución. Por tanto,

la técnica de *Monitorización Dual* no requiere almacenar información calculada en tiempo de compilación, lo que se traduce en que los recursos necesarios para su implementación son reducidos y el impacto sobre el rendimiento del sistema es pequeño. Además, los dos puntos de observación se sitúan justo en la entrada y en la salida del flujo de instrucciones en el microprocesador, lo que posibilita la detección de cualquier error que se produzca en las etapas intermedias del *pipeline*.

3.4.1. Puntos de observación

La observabilidad del sistema se aumenta observando el comportamiento de la ejecución desde dos puntos distintos. En la **¡Error! No se encuentra el origen de la referencia.** se muestran los dos puntos de observación utilizados en esta técnica.

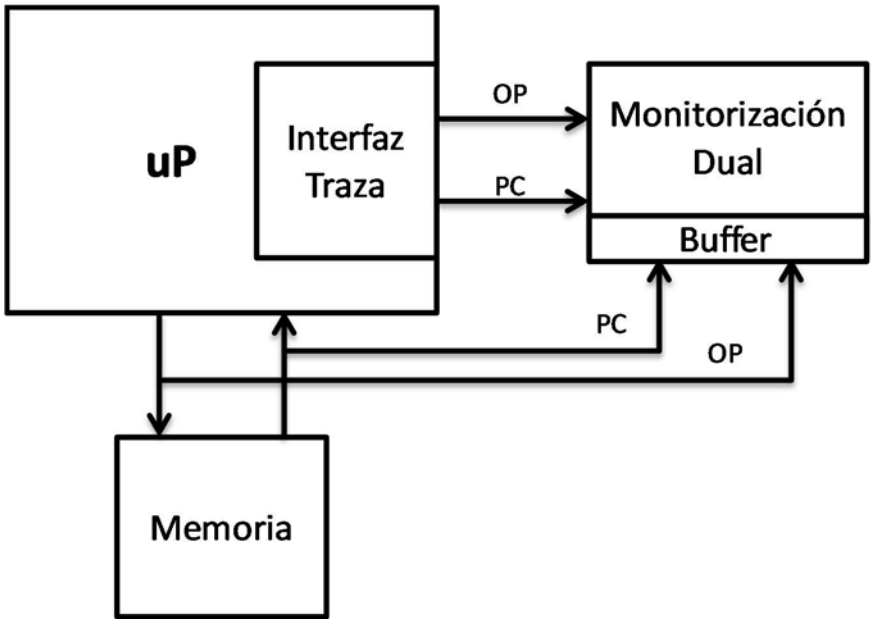

Figura 13: Puntos de observación de la técnica Monitorización Dual.

El primero de ellos es el bus entre el microprocesador y la memoria de instrucciones. Este es el punto de observación que se ha utilizado tradicionalmente, el cual proporciona el valor del contador de programa (PC) y el código de cada instrucción en la etapa de búsqueda, justo en el momento en el que la instrucción se carga en el microprocesador. El segundo punto de observación es la *interfaz de traza*, la cual proporciona la información más relevante de cada instrucción ejecutada, incluyendo

también el contador de programa y el código de instrucción, pero después de que la instrucción haya sido ejecutada. Se puede acceder a la *interfaz de traza* sin afectar a la ejecución del microprocesador y sin afectar al rendimiento del mismo. Además el uso de la *interfaz de traza* como punto de observación no interfiere en un posible uso para tareas de depuración.

Una vez que una instrucción se carga desde la memoria, la información de dicha instrucción viaja a través de la ruta de datos del procesador y se utiliza en cada etapa para gestionar las diferentes operaciones que realiza el microprocesador durante la ejecución de la instrucción. El principio básico por el que funciona la Monitorización Dual es que si en alguna etapa se produjera un error en el PC o en el registro de instrucción (IR), este error se propagaría hasta la interfaz de traza y se detectaría al comparar los valores obtenidos tras la ejecución con los que se capturaron aguas arriba cuando la instrucción entró en el procesador. Es importante destacar que este mecanismo de detección no precisa de ninguna información adicional para realizar la comparación, ya que toda la información necesaria se obtiene de la propia ejecución.

Para sincronizar la información capturada en la interfaz de traza con la del bus de memoria se introduce una pequeña memoria intermedia (*buffer*) de un tamaño equivalente al número de etapas del procesador. Esta memoria almacena el PC y el IR de cada instrucción que aparece en el bus, los cuales se comparan con los valores correspondientes que se obtienen por la interfaz de traza en orden de aparición.

Si se produce un error en el PC durante la etapa de búsqueda de la instrucción, éste podría no ser detectado, puesto que la dirección de la instrucción es generada por el propio procesador. En tal caso ambos puntos de monitorización proporcionarían la misma información errónea. Esto es debido a que el error se ha producido en un instante de la ejecución anterior a los dos puntos de observación. Este tipo de errores se detectan de una manera eficiente mediante la técnica de Predicción del Contador de Programa, descrita en el apartado 3.2.

3.4.2. Implementación de la técnica *de Monitorización Dual*

La técnica de *Monitorización Dual* ha sido implementada en un módulo hardware externo con la finalidad de comprobar la eficiencia de la misma para detectar errores de flujo. En la Figura 14 se muestra la estructura interna de un módulo hardware encargado de implementar la técnica de *Monitorización Dual*.

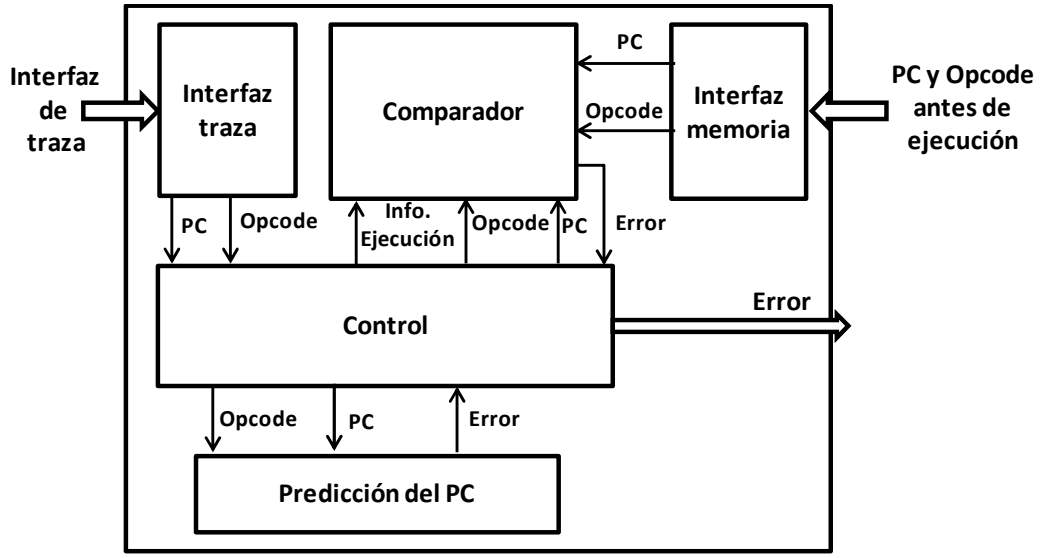

Figura 14: Esquema de la técnica de Monitorización Dual.

A continuación se describe brevemente la funcionalidad de cada uno de los bloques que forman parte del módulo:

- *Interfaz traza*: Bloque encargado de captar la información más relevante de la ejecución del microprocesador, a través de la *interfaz de traza*, y transferir dicha información al bloque de *Control*. La información de la ejecución que se utiliza en esta técnica es el contador de programa y el código de instrucción.
- *Interfaz de memoria*: Bloque encargado de capturar del bus de caché o bus de memoria el valor del contador de programa y el código de instrucción de la siguiente instrucción que va a ser ejecutada. Los valores del contador de programa y el código de instrucción se transfieren al bloque *Comparador* para que pueda compararlos con los valores proporcionados por el *Control*.

- *Control*: Máquina de estados encargada de coordinar el funcionamiento de los distintos bloques del módulo. Las funciones principales del control son las siguientes:
    - Gestionar el funcionamiento del módulo teniendo en cuenta el valor del contador de programa y del código de instrucción proporcionado por la interfaz de traza.
    - Indicar al bloque *Comparador* cuando se produce la llegada de una nueva instrucción a través de la *interfaz de traza*, y proporcionar los valores del contador de programa y el código de instrucción.
    - Indica cuando debe de realizarse una nueva comparación.
    - Indica si ha habido un error en el flujo de programa a través de una señal de error.
- *Comparador*: bloque encargado de comparar los valores del contador de programa y del código de instrucción obtenidos de la *interfaz de traza* y del bus de memoria, es decir, se encarga de comparar los valores del contador de programa y del código de instrucción antes y después de la ejecución de cada instrucción. Este bloque cuenta con un buffer de tamaño igual al número de etapas de pipeline del procesador, y almacena la información obtenida a través del bus la memoria para sincronizarla con la información proporcionada a través de la *interfaz de traza*. Para implementar dicho buffer se ha utilizado una pequeña memoria *FIFO* (*First In First* Out) del tamaño del *pipeline*.
- *Predicción del PC*: Bloque encargado de calcular el valor del contador de programa para la instrucción siguiente y compararlo cuando ésta llega, de acuerdo con la técnica descrita en el apartado 3.2.

La implementación permite aprovechar también información adicional proporcionada desde la interfaz de traza. Además de la comparación de las instrucciones y de la predicción del PC, el módulo de control también puede comprobar la etiqueta temporal y los bits de interrupción y de error. La etiqueta temporal es la salida de un contador interno del procesador que

se incrementa cada ciclo de reloj mientras el procesador está corriendo. El monitor incluye una condición de tiempo límite (*timeout*) para detectar un posible bloqueo del procesador inducido por un fallo. Esta condición genera una señal de error en el caso de que no se generen nuevas instrucciones durante un periodo de tiempo dado.

Los bits de interrupción y error se usan para tratar las excepciones. Es importante distinguir entre interrupciones inesperadas, inducidas por fallos, e interrupciones implementadas, que pueden ocurrir en el funcionamiento normal. Las interrupciones inesperadas se pueden producir, por ejemplo, cuando un fallo produce una instrucción no válida o una dirección de memoria no válida. Estas interrupciones se pueden distinguir de las interrupciones implementadas comprobando la siguiente instrucción proporcionada por la interfaz de traza. Cuando ocurre una interrupción inesperada o el procesador entra en modo de error se activa la señal de error mostrada en la Figura 12. Alternativamente, se pueden cubrir las interrupciones inesperadas mediante rutinas de atención a la interrupción apropiadas.

3.4.3. Análisis de la técnica de Monitorización Dual

La técnica de *Monitorización Dual* se basa principalmente en aumentar la observabilidad del sistema gracias a la utilización de dos puntos de observación diferentes. La información obtenida a través de ambos puntos de observación es comparada con la finalidad de buscar errores que se hayan producido durante la ejecución de las diferentes etapas del *pipeline*. Gracias al aumento de observabilidad del sistema no es necesario almacenar información de la ejecución calculada en tiempo de compilación, ya que la información necesaria para realizar las comparaciones se puede obtener directamente en tiempo de ejecución. En consecuencia, tampoco es necesario llevar a cabo un análisis previo del software a ejecutar, y por tanto el tiempo de *setup* del sistema se ve considerablemente reducido. Todo esto se traduce en una reducción significativa de los recursos necesarios para implementar la técnica y finalmente en una reducción del impacto sobre el rendimiento del sistema.

Como se demostrará en los capítulos siguientes, la técnica *Monitorización Dual* es capaz de detectar el 100% de los errores que afectan al flujo de programa. Todo error que produzca una modificación en el contador de programa o en el código de instrucción en cualquiera de las etapas del *pipeline* es detectado. Además, la latencia de detección de errores es muy baja, igual al tiempo necesario para ejecutar cada una de las etapas del pipeline de una instrucción en el peor caso.

Una de las principales limitaciones que presenta la técnica *Monitorización Dual* es que no puede detectar aquellos errores que se producen en un instante temporal previo a ambos puntos de observación. En esta situación la información que se obtiene a través de ambos puntos de observación es la misma y, por tanto, al llevar a cabo la comparación no se consigue detectar el error. Este tipo de errores se producen principalmente cuando el fallo afecta al contador de programa. Sin embargo, si se combina con la técnica de Predicción del Contador de Programa, este tipo de errores se pueden detectar

3.5. Resumen y conclusiones

Las técnicas de *CFC* nos permiten detectar errores en el flujo de programa de una manera eficiente. La mayoría de las técnicas de *CFC* propuestas por otros autores precisan calcular la información de la ejecución en tiempo de compilación y almacenarla para posteriormente compararla con la ejecución online [Mich91], [Berg09], o realizar una duplicación de la ejecución [Gros10], Esto produce un impacto notable sobre los recursos necesarios y el rendimiento del sistema. Las técnicas de *CFC* presentadas en este capítulo buscan alcanzar una solución de compromiso entre la tolerancia a fallos y el impacto sobre el rendimiento del sistema, intentando mejorar este compromiso respecto a las técnicas existentes. Así mismo, se ha buscado desarrollar técnicas no intrusivas, con el menor coste posible y buscando siempre que la latencia de detección sea la menor posible.

Para conseguir estos objetivos, un elemento fundamental es la utilización de la *interfaz de traza* como punto de observación de la ejecución. La *interfaz de traza* permite observar el flujo de instrucciones del microprocesador de una manera no intrusiva. Las infraestructuras de depuración, como la interfaz de traza, están principalmente concebidas para ser utilizadas durante la fase de desarrollo del sistema. Puesto que no se utilizan durante la fase de operación, se pueden reutilizar para la detección de errores. Por otra parte, al utilizar la *interfaz de traza* como punto de observación se está reutilizando un hardware existente previamente, por lo que el impacto sobre el sistema es mínimo. De esta manera se consigue aumentar la observabilidad del sistema a muy bajo coste.

En este capítulo se han descrito tres técnicas de CFC que son aportación original de esta tesis doctoral proporcionando un amplio abanico de soluciones adaptable según las necesidades del microprocesador utilizado y el software a ejecutar.

La técnica de *Predicción del contador del programa* calcula el valor del contador de programa teniendo en cuenta el valor anterior del mismo y el código de instrucción actual. Los valores tanto del contador de programa como del código de instrucción se obtienen a través de la *interfaz de traza*, presente en la mayoría de los microprocesadores modernos. La información proporcionada a través de la *interfaz de traza* se corresponde con la información de la instrucción que acaba de ejecutar el microprocesador. Por tanto, esta técnica puede detectar errores en todas las etapas del pipeline, siempre que afecten al contador de programa o al código de instrucción.

La técnica de *Monitorización de Firmas* verifica la ejecución de un bloque de instrucciones mediante el cálculo de una firma *online*. Cada bloque tiene asignada una firma de referencia que es calculada en tiempo de compilación. Al final de la ejecución de cada bloque se comparan la firma de referencia con la calculada durante la ejecución para determinar si se ha producido un error en la ejecución de dicho bloque. Esta técnica

presenta dos mejoras respecto a técnicas basadas en el cálculo de firmas propuestas por otros autores. Pon un lado se consigue reducir el tamaño de la memoria necesaria para almacenar las firmas de referencia utilizando la tabla *CFC-ST*. Se han propuesto dos métodos de almacenamiento y acceso a la tabla *CFC-ST,* un método estático y un método dinámico. En ambos casos se consigue reducir el impacto sobre el rendimiento del sistema al reducir el tamaño de la tabla necesaria para almacenar las firmas. Por otro lado, gracias a la mejora de *Address Checking* se detecta un porcentaje de errores mayor, ya que se pueden detectar aquellos errores que provocan una pérdida de secuencia en la ejecución, y además se minimiza la latencia de detección para aquellos errores que provocan un salto ilegal dentro de un bloque básico. Al igual que la técnica de *Monitorización Dual*, esta técnica puede ser implementada junto con la técnica de *Predicción del contador de programa* complementándose en la detección de errores de una manera eficiente.

La técnica de *Monitorización Dual* realiza la monitorización de la ejecución utilizando dos puntos de observación diferentes, la interfaz de traza y el bus de memoria. A través de estos dos puntos de observación se obtienen el contador de programa y el código de instrucción en la etapa de búsqueda y justo después de la etapa de ejecución, permitiendo la comparación de estos valores antes y después de que se haya ejecutado cada instrucción. Al igual que en la técnica de Predicción del Contador de Programa, si se produce un error de flujo que afecte tanto al contador de programa como al código de instrucción en cualquiera de las etapas del pipeline, éste será detectado. Mediante la combinación con la técnica de Predicción del Contador de Programa se pueden detectar además errores que se produzcan en el contador de programa en la etapa de búsqueda y que podrían afectar a ambos puntos de observación

Las técnicas de *Predicción del Contador de Programa* y *Monitorización Dual* consiguen eliminar la necesidad de almacenar información calculada en tiempo de compilación, y por tanto, reducen de una manera drástica los recursos necesarios para su implementación y el impacto sobre el

rendimiento del sistema. Así mismo, consiguen alcanzar una cobertura de detección de fallos del 100% si únicamente se tienen en cuenta aquellos errores que afectan al flujo de programa. También es importante destacar que se consiguen alcanzar porcentajes de detección elevados cuando se tienen en cuenta todos los posibles errores, y no únicamente los que afectan al flujo de programa. Un claro ejemplo de esta capacidad de detección se aprecia en los resultados expuestos en los apartados 4.3.3.3 y 4.5.2 del siguiente capítulo. En estos experimentos se alcanza un porcentaje de detección de errores del 70% implementando las técnicas *Predicción del contador de programa* y *Monitorización Dual,* mejorándose este porcentaje hasta el 92% cuando ambas técnicas son implementadas en combinación con técnicas de endurecimiento software, las cuales nos permiten detectar errores que afectan a los datos.

CAPÍTULO 4

RESULTADOS EXPERIMENTALES DE LAS TÉCNICAS DE CONTROL DEL FLUJO DE PROGRAMA

4. Resultados experimentales de las técnicas de control del flujo de programa

En este capítulo se muestran los resultados experimentales obtenidos con las técnicas propuestas en esta tesis que se han descrito en el capítulo 3. Para ello se han realizado diversas campañas de inyección de fallos mediante simulación y emulación en FPGA.

El objetivo de los experimentos es validar las técnicas y determinar su capacidad de detección de errores. Los resultados obtenidos demuestran que las técnicas propuestas son efectivas y mejoran los resultados de las soluciones existentes.

Los experimentos se han realizado con dos microprocesadores de características distintas, concretamente LEON3 y PicoBlaze. En el primer apartado se resumen las principales características de estos microprocesadores. También se han utilizado diferentes aplicaciones software que se detallan en cada experimento.

4.1. Casos de estudio

Para la validación de las técnicas se han utilizado dos microprocesadores de diferente complejidad, *PicoBlaze y LEON3*. En los siguientes apartados se describen las principales características de ambos microprocesadores.

4.1.1. Microprocesador PicoBlaze

EL microprocesador *PicoBlaze* [Xili11] es un *soft-core* de 8 bits con una arquitectura *RISC*, especialmente diseñado para su uso en sistemas empotrados basados en FPGA. Es un microprocesador sencillo, con grandes limitaciones en cuanto a rendimiento y recursos, pero que tiene un conjunto de instrucciones relativamente completo. Una de sus principales características es que tiene una capacidad de direccionamiento muy pequeña, tanto en datos como en programa. Esta característica hace que las instrucciones sean pequeñas y que el microprocesador ocupe pocos

recursos, por lo que está indicado para aplicaciones sencillas de control (microcontrolador) de bajas prestaciones, en las que puede ser muy eficiente.

De forma más detallada, las características principales del microprocesador *PicoBlaze* son las siguientes (Figura 15):

- 16 registros de propósito general de 8 bits cada uno. PicoBlaze tiene una arquitectura simple en la cual no hay registros reservados para operaciones especiales y tampoco hay registros con prioridad sobre otros.
- Memoria de programa de hasta 1024 instrucciones almacenadas en una memoria interna, normalmente implementada con un bloque de RAM de la FPGA. Cada instrucción tiene un ancho de 18 bits. Las instrucciones se cargan durante el proceso de configuración de la FPGA.
- Unidad aritmético-lógica de 8 bits con bit de acarreo. La unidad aritmético-lógica realiza todas las operaciones del microprocesador, incluyendo:
    - Operaciones aritméticas básicas como la suma y la resta.
    - Operaciones lógicas bit a bit como *AND, OR y XOR*.
    - Comparaciones aritméticas.
    - Operaciones de desplazamiento y rotación.
- Scratchpad RAM de 64 bytes de propósito general, direccionable de manera directa o indirecta, utilizando las instrucciones Store y Fetch.
- 256 puertos de entrada y 256 puertos de salida. Estos puertos permiten al microprocesador Picoblaze interconectarse con otros periféricos o con lógica de la FPGA, aumentando por tanto sus capacidades.
- Pila de 31 posiciones, lo que permite 31 llamadas a subrutina anidadas.
- Interrupciones. El microprocesador PicoBlaze tiene una entrada opcional denominada *INTERRUPT*, que permite al procesador controlar eventos externos. La respuesta del microprocesador a una interrupción se realiza rápidamente, en sólo 5 ciclos de reloj.

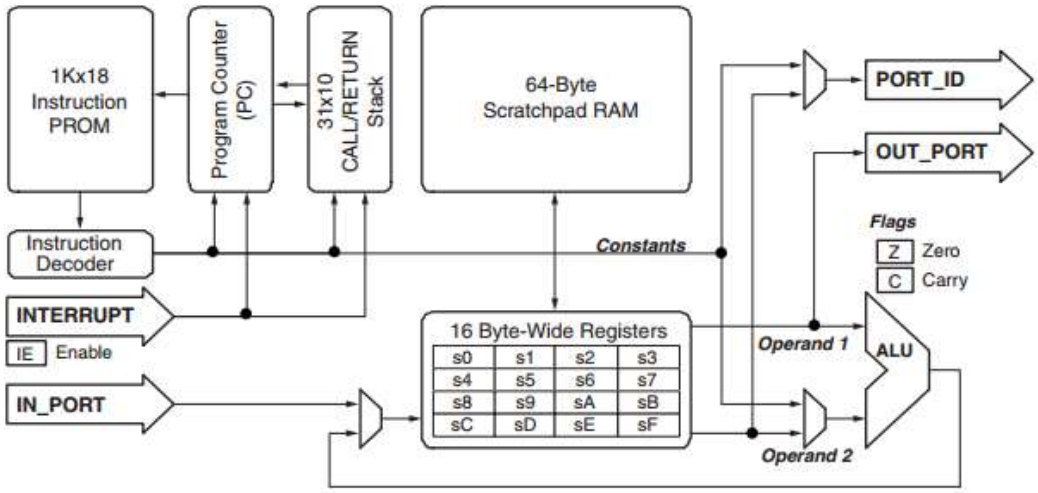

Figura 15: Diagrama de bloques del microprocesador PicoBlaze [Xili11].

Picoblaze es un microprocesador diseñado exclusivamente para FPGAs de Xilinx, lo que resulta muy específico. Además, la tecnología FPGA plantea problemas de sensibilidad a errores muy diferentes a los de una tecnología ASIC. Por este motivo, en los experimentos se ha utilizado un diseño equivalente a nivel de transferencia entre registros (RTLPicoBlaze) obtenido a partir de la versión original de Picoblaze-3. Estes diseño se sintetizó posteriormente para una tecnología ASIC de 90 nm, utilizando la biblioteca SAED90 proporcionada por Synopsys.

La versión original de Picoblaze no dispone de interfaz de traza, dado que se trata de un microprocesador muy sencillo. Por este motivo, el diseño del microprocesador se ha extendido con una interfaz de traza que proporciona la información necesaria (contador de programa y código de instrucción) similar a la que se puede encontrar en el microprocesador LEON3. Esta interfaz es necesaria para poder conectar posteriormente un monitor hardware externo.

4.1.2. Microprocesador *LEON3*

El *LEON3* [Gais06] es un microprocesador sintetizable de 32 bits con arquitectura *SPARC V8,* de amplio uso en aplicaciones espaciales. Está principalmente diseñado para aplicaciones empotradas, las cuales

combinan un alto rendimiento con una complejidad y un consumo de energía bajos. El microprocesador *LEON3* cuenta con una unidad de procesamiento de enteros con 7 etapas de *pipeline,* arquitectura *Hardvard* (caché de instrucciones y de datos separadas), multiplicador y divisor hardware, y sistema integrado de soporte a la depuración. Se distribuye como parte de la librería *GRLIB IP* [Gais10]. La librería *GRLIB IP* engloba los diseños de diversos componentes para el desarrollo de sistemas empotrados.

La arquitectura de un sistema basado en el microprocesador *LEON3* consiste principalmente en el microprocesador y un conjunto de módulos IP conectados a través de los buses *AMBA (Advanced Microcontroller Bus Architecture) AHB* (AMBA High-performance Bus) y *AMBA APB (Advanced Peripheral Bus)*, tal y como se muestra en la Figura 16.

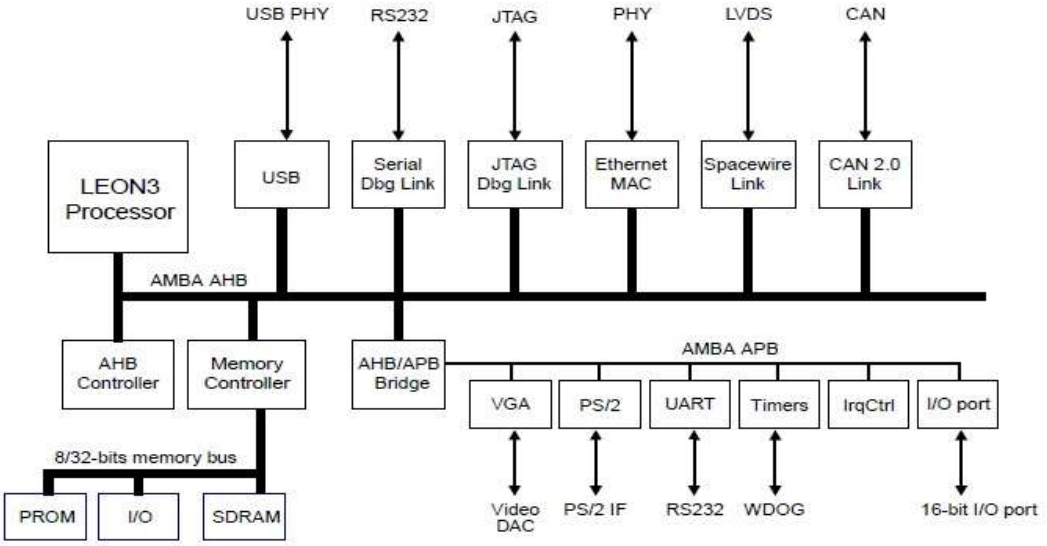

Figura 16: Arquitectura de un sistema basado en el microprocesador LEON3 [Gais10].

El sistema se centra en torno al bus de sistema *AMBA AHB*, al que se conectan el microprocesador *LEON3*, los controladores de memoria y del bus, así como otros dispositivos de elevado ancho de banda. El sistema permite utilizar una amplia gama de periféricos, entre los que se pueden destacar los siguientes: *USB*, *CAN-2.0*, *SpaceWire*, *RS232, JTAG*, *UART*, *VGA*, *PS/2*, temporizadores, controlador de interrupciones y puertos de entrada/salida. El bus *AHB* se comunica con el bus *APB* a través del puente

(*bridge*) *AHB*/*APB*, el cual es el maestro del bus *APB*. Al bus *APB* se conectan periféricos de menor ancho de banda como la *UART*, *GPIO*, *VGA*, temporizadores, *PS/2*, *etc.*.

La Figura 17 muestra la estructura interna del núcleo del microprocesador, la unidad de enteros, la cual tiene las siguientes características principales:

- Segmentación en 7 etapas: búsqueda, decodificación, acceso a registros, ejecución, acceso a memoria, excepción y escritura.
- Caché de instrucciones y de datos separadas.
- Hardware de multiplicación y división.
- Soporte de 2 a 32 ventanas de registros
- Hardware breakpoint: registros que comparan su valor continuamente con el contador de programa, provocando una interrupción si alguno de ellos coincide.
- Interfaz y buffer de traza: buffer circular que almacena las instrucciones ejecutadas.

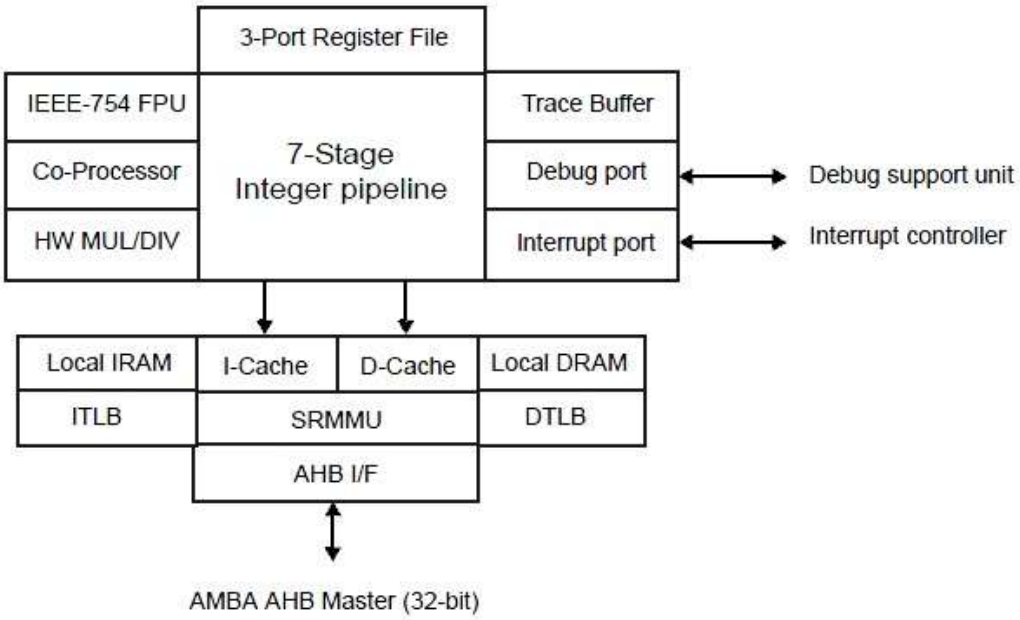

Figura 17: Integer Unit del microprocesador LEON3.

En esta tesis se ha reutilizado la interfaz de traza como punto de observación para implementar las técnicas descritas en el capítulo 3.

4.2. Inyección de fallos

Para la validación de técnicas de detección de errores se utiliza habitualmente la inyección de fallos, la cual se puede realizar mediante simulación o emulación con FPGAs. En esta tesis se han utilizado ambas técnicas.

La simulación es generalmente más sencilla de implementar, ya que puede realizarse con la ayuda de herramientas comerciales de simulación, y es la solución generalmente adoptada en la mayoría de los trabajos publicados sobre fallos en microprocesadores. En concreto, para los experimentos basados en simulación que se han realizado en esta tesis se ha utilizado la herramienta *Modelsim* de *Mentor Graphics*, la cual permite inyectar fallos en cualquier señal de un diseño y en cualquier instante.

Entre las ventajas de la inyección de fallos mediante simulación cabe destacar que se pueden implementar modelos de fallos muy variados. Por otra parte, facilita la comparación de resultados con otras técnicas, puesto que la mayoría de los trabajos publicados sobre fallos en microprocesadores están validados mediante simulación. Sin embargo, la inyección de fallos mediante simulación presenta algunos inconvenientes importantes. La simulación es generalmente muy lenta, lo que en la práctica impide realizar campañas de inyección de fallos muy grandes. Este inconveniente es particularmente relevante para el caso de fallos de tipo SET, puesto que éstos pueden producirse en cualquier instante y en cualquier nodo del circuito, y se deben tener en cuenta los retardos de propagación. Por este motivo, en los experimentos de simulación se inyectan principalmente SEUs, debido a la complejidad de los modelos de fallos de tipo SET y al importante tiempo de ejecución que conllevan,

La emulación con FPGAs permite acelerar notablemente la inyección de fallos y realizar campañas de inyección de fallos masivas. En esta tesis, los experimentos de emulación se han realizado con la herramienta AMUSE (*Autonomous MUltilevel emulation system for Soft Error evaluation*) [Entr12]. AMUSE es un sistema de emulación que integra eficientemente el modelo a

nivel puerta y a nivel RT de un circuito bajo test en una FPGA. El cambio entre estos modelos se puede realizar en cualquier momento durante la emulación, de manera que la inyección del fallo se realiza de manera precisa en el nivel de puerta, mientras que la propagación del fallo en sucesivos ciclos de reloj es realizada en el nivel RT consiguiendo una ejecución más rápida. Este mecanismo está especialmente diseñado para soportar la inyección de SETs con suficiente precisión y una alta velocidad. El incremento en el área que se produce al introducir el nivel RT es ampliamente compensado con la mejora en el rendimiento, ya que el modelo RT se ejecuta dos órdenes de magnitud más rápido que el modelo a nivel puerta. Gracias a este enfoque, se pueden realizar amplias campañas de inyección de errores en un tiempo aceptable.

La inyección de fallos tiene por objeto clasificar los efectos de los fallos. Habitualmente se consideran los siguientes casos:

- Silencioso: el fallo inyectado no ha producido ningún error en la ejecución.
- Error: el fallo inyectado ha producido error en la ejecución. A su vez los errores han sido clasificados de acuerdo con el efecto que producían en el comportamiento del programa, como se propone en [Mukh03].
    - Fallos tipo *Silent Data Corruption (SDC):* son fallos que no han sido detectados o corregidos y en los que el programa finaliza con un resultado erróneo.
    - Fallos tipo *HANG*: son los fallos que producen una terminación anormal del programa o un bucle infinito. Para detectar este tipo de error se establece un tiempo de espera, al cabo del cual, si no ha finalizado correctamente la ejecución se considera que se ha producido un error.
- Detectado: el fallo ha sido detectado mediante algún mecanismo de detección, si se ha implementado.

4.3. Análisis de la Predicción del Contador de Programa en el microprocesador Picoblaze

Para realizar un análisis de la capacidad de detección de errores de la técnica de *Predicción del Contador de Programa* se realizó en primer lugar una campaña de inyección de fallos utilizando un sistema basado en el microprocesador *PicoBlaze* como caso de estudio. La razón por la que se eligió este microprocesador es que, dada la menor complejidad de su arquitectura, se puede analizar la eficacia de la técnica de una manera más sencilla que si los experimentos se realizaran directamente sobre un microprocesador complejo.

En este apartado se muestran los experimentos realizados para la técnica de *Predicción del Contador de Programa* utilizando el microprocesador *PicoBlaze* como caso de estudio, detallando el método experimental utilizado, la estructura interna del *monitor hardware* que implementa la técnica, así como los resultados experimentales obtenidos. Estos resultados han sido publicados en [Parr14a], en colaboración con la Universidad de Alicante.

4.3.1. Descripción del Monitor Hardware utilizado

La técnica de *Predicción del contador de programa* ha sido implementada en un módulo hardware externo, al que se ha denominado *Monitor Hardware*, para poder ser integrado en un sistema basado en el microprocesador *PicoBlaze*.

La arquitectura del Monitor Hardware utilizado es la misma que se mostró en el capítulo 3 (Figura 3). No obstante, se ha extendido esta arquitectura con una réplica de la pila de un número de niveles (N) configurable. Esta extensión es necesaria para poder realizar el control del flujo de programa cuando se produzcan llamadas a subrutina, puesto que Picoblaze utiliza una pila que es independiente de la memoria. La réplica de la pila debe ser configurada con un número de niveles acorde con el máximo nivel de anidamiento del programa que se vaya a ejecutar. En la práctica este valor

es generalmente inferior a los 31 niveles de que consta la pila de PicoBlaze, lo que permite optimizar los recursos.

El *Monitor Hardware* tiene en cuenta si una instrucción es de salto o no a la hora de predecir el valor del contador de programa, distinguiendo entre saltos condicionales e incondicionales. Cuando se produce una llamada a subrutina, el valor de la dirección de retorno es almacenado en la réplica de la pila. En el momento que la instrucción de retorno es ejecutada, el valor del contador de programa estimado se calcula recuperando el valor almacenado en la pila. La configuración del número de niveles implementados en la réplica de la pila puede basarse en una solución de compromiso entre los recursos necesarios y la cobertura de errores. En aplicaciones con bajo nivel de subrutinas anidadas basta con replicar unas pocas posiciones de la pila. En la práctica puede ser admisible que el número de subrutinas anidadas sea superior al tamaño de la pila replicada, a costa de no poder detectar algunos errores para niveles altos de anidamiento. En los experimentos se han replicado 3 posiciones de la pila, lo que es suficiente para los programas de prueba utilizados

4.3.2. Características de los experimentos

Para comprobar la capacidad para detectar errores de flujo de la técnica *Predicción del contador de programa* se han llevado a cabo experimentos de emulación en FPGA realizando una serie de campañas de inyección de fallos de tipo *SEU* y *SET*. Para la realización de los experimentos se ha utilizado la herramienta AMUSE [Entr12]. Gracias a las altas prestaciones de la herramienta de inyección AMUSE se han podido inyectar millones de fallos en un tiempo reducido y, por tanto, realizar una evaluación exhaustiva de la técnica.

En los experimentos de inyección de fallos de tipo *SEU* se inyectaron fallos de forma exhaustiva en todos los registros del microprocesador en cada uno los ciclos de reloj. En los experimentos de inyección de fallos tipo *SET* se inyectaron fallos en todas las puertas del microprocesador en instantes aleatorios dentro de cada ciclo de reloj. Se eligió un ancho de pulso de 500

ps, que es aproximadamente el 20% del periodo de reloj. Para cada puerta se generó una lista de instantes de inyección con un intervalo de tiempo aleatorio entre dos instantes de inyección consecutivos. La media de este intervalo de tiempo aleatorio es de 500 ps (igual que el ancho de pulso del fallo). El periodo de reloj es de 2,67 ns, y por tanto, se inyectan de media 5,3 fallos en cada ciclo de reloj. . De esta manera se garantiza que se cubren todos los ciclos de reloj y con un número similar de fallos.

Se han utilizado tres aplicaciones software diferentes para la realización de los experimentos:

- Multiplicación de matrices (Mmult)
- Controlador Proporcional-Integral-Derivativo (PID)
- Filtro de respuesta finita al impulso (FIR).

En la Tabla 3 se muestra el tamaño de las aplicaciones utilizadas. El número total de fallos inyectados varía para aplicación software. En total, se inyectaron hasta 20 millones de fallos de tipo *SEU* y 111 millones de fallos de tipo *SET*.

Aplicación Sofware	***Tamaño (ciclos de reloj)***
Mmult	3304
PID	7390
FIR	7374

Tabla 3: Tamaño de las aplicaciones software utilizadas

Los fallos han sido clasificados de acuerdo con la clasificación propuesta en el apartado de introducción de este capítulo. En el caso de los fallos tipo *HANG*, el número de ciclos del tiempo de espera que se ha establecido depende de la aplicación software, teniendo un rango entre 100 y 3300 ciclos de reloj para los experimentos realizados.

4.3.3. Resultados experimentales

En la Tabla 4 se muestran los resultados obtenidos en la síntesis del microprocesador PicoBlaze y el Monitor Hardware para una tecnología de 90 nm utilizando la librería SAED90 nm proporcionada por Synopsys [SAED90]. El *Monitor Hardware* implica un incremento de 435 puertas y 119 registros, que se corresponde con un incremento del área de un 40% aproximadamente respecto al microprocesador *PicoBlaze*. Este incremento es importante en en relación al tamaño del microprocesador, como consecuencia de que éste es muy pequeño. No obstante, para microprocesadores más complejos el incremento de tamaño esperado es mucho menor. El *Monitor Hardware* no afecta a la frecuencia máxima de operación del procesador.

	#Puertas	***#Biestables***	***Frecuencia máxima (MHz)***
Microprocesador	1066	194	350
Monitor Hardware	435	119	670

Tabla 4: Resultados de síntesis del diseño basado en el microprocesador PicoBlaze y HM-Predicción del PC.

En la Tabla 5 y la Tabla 6 se muestran los resultados experimentales para las campañas de inyección de fallos de tipo *SEU* y *SET,* respectivamente, para las tres aplicaciones software utilizadas. Las dos primeras columnas muestran la aplicación software y el tipo de error, la tercera columna muestra los resultados obtenidos sin incluir el *Monitor Hardware*, mientras que la cuarta columna muestra los resultados obtenidos cuanto se utiliza el Monitor Hardware. Como se puede observar en estos resultados, el *Monitor Hardware* es capaz de detectar la mayoría de los errores de tipo HANG y una fracción menor de los errores de tipo SDC. Esto es debido a que la técnica de *Predicción del contador de programa* únicamente puede detectar aquellos errores que afectan al contador de programa. La mayoría de estos errores hacen que se pierda la secuencia de ejecución, lo que se traduce en una pérdida de secuencia que se manifiesta como un error de tipo *HANG*. En cambio, los errores de tipo *SDC* están generalmente

asociados a los datos y en mucha menor medida al flujo de programa, por lo que la mayoría de estos errores no se pueden detectar con esta técnica. Los errores de tipo SDC que son detectados mediante la *Predicción del contador de programa* son aquellos que provocan un cambio incorrecto en la secuencia de ejecución, afectando a los cómputos, pero permitiendo que el programa termine.

Los resultados demuestran la relativa importancia de los errores que afectan al contador de programa, que son los que esta técnica puede detectar. En conjunto, se detectan entre un 40% y un 49% de los fallos de tipo SEU, y entre un 60% y un 90% de los fallos de tipo SET.

Software	**Tipo de error**	**Sin *HM***	**Con *HM***
Mmult	***SDC(%)***	9,65%	8,43%
	HANG(%)	6,94%	0,00%
	Total(%)	16,59%	8,43%
PID	***SDC(%)***	12,97%	10,51%
	HANG(%)	7,77%	1,95%
	Total(%)	20,74%	12,47%
FIR	***SDC(%)***	12,39%	9,91%
	HANG(%)	5,55%	0,01%
	Total(%)	17,94%	9,91%

Tabla 5: Campaña de inyección de fallos SEU en emulación para el sistema basado en el microprocesador PicoBlaze y HM-Predicción del PC.

Software	**Tipo de error**	**Sin *HM***	**Con *HM***
Mmult	***SDC (%)***	0,83%	0,17%
	HANG(%)	0,90%	0,00%
	Total(%)	1,73%	0,17%
PID	***SDC (%)***	1,30%	0,85%
	HANG(%)	1,18%	0,14%
	Total(%)	2,48%	0,98%
FIR	***SDC (%)***	1,20%	0,74%
	HANG(%)	1,04%	0,00%

	Total(%)	2,24%	0,74%

Tabla 6: Campaña de inyección de fallos SET en emulación para el sistema basado en el microprocesador PicoBlaze y HM-Predicción del PC.

Para conseguir una mayor robustez, la técnica de Predicción del contador de programa se puede combinar con técnicas de software. Los resultados para este caso se analizan en el capítulo 5.

4.4. Análisis de la Monitorización de Firmas en el microprocesador LEON3

Para validar la técnica de Monitorización de Firmas se han realizado campañas de inyección de fallos mediante simulación con el microprocesador LEON3. En primer lugar, se han realizado unos primeros experimentos sencillos con la idea de analizar la necesidad de combinar esta técnica con otras como la Predicción del Contador de Programa o el *Address Checking*. Los resultados de estos primeros experimentos se han publicado en [Parr11].

Como se ha señalado en el capítulo anterior, la Monitorización de firmas por sí sola puede resultar insuficiente, debido a que las firmas solo se comprueban al finalizar la ejecución de un bloque básico. Si no se detecta el final de un bloque básico por error, entonces la comprobación no se efectúa y se puede perder el control de la ejecución. Para corregir este problema es importante combinar la Monitorización de Firmas con otras técnicas.

Tras estos primeros experimentos, se ha procedido a realizar una campaña de inyección de fallos de mayor envergadura. Para esta prueba se han combinado las técnicas *Monitorización de firmas y Predicción del Contador de Programa* en el *Monitor Hardware*. Se han realizado experimentos más amplios, utilizando diferentes códigos y tres modelos de fallo diferentes. Con estos experimentos se busca principalmente corroborar los buenos resultados obtenidos en los primeros experimentos de simulación, analizando el comportamiento de ambas técnicas para diferentes aplicaciones software y centrándonos en la inyección de fallos que a priori

únicamente afecten al flujo de programa. Los resultados obtenidos en esta campaña de inyección han sido publicados en [Boya15].

4.4.1. Primeros experimentos

En primer lugar se han realizado una serie de experimentos de simulación con el objetivo de explorar el potencial de las diferentes técnicas propuestas en relación con la Monitorización de Firmas y la Predicción del Contador de Programa. Una de las ventajas de realizar experimentos de simulación es que es posible saber qué está sucediendo en la ejecución con tan solo observar la forma de onda proporcionada a través del programa que realiza la simulación, en este caso *Modelsim*. De esta manera se puede saber exactamente qué tipos de errores son detectados y cuáles no, y el motivo por el que no son detectados. Esto permite determinar si las técnicas están funcionando correctamente y también se obtiene información muy valiosa acerca de cómo mejorarlas.

Los resultados experimentales obtenidos en esta campaña de inyección han sido publicados en [Parr11].

Para estos primeros experimentos se ha implementado un *Monitor Hardware* que incluye la técnica de *Predicción del Contador de Programa* y una versión convencional de la *Monitorización de Firmas*. El *Monitor Hardware* se ha conectado a la *interfaz de traza* para poder llevar a cabo la observación de la ejecución y obtener la información necesaria para implementar ambas técnicas.

Gracias a la información obtenida a través de la *Interfaz de traza* se puede llevar a cabo también un tratamiento de las excepciones utilizando la información que proporciona el *flag de interrupciones* y el *flag de error*. Es importante diferenciar entre excepciones implementadas, que se esperan que ocurran durante una ejecución normal, y excepciones debidas a un fallo, las cuales no son esperadas durante la ejecución. Para las excepciones implementadas, el *Monitor Hardware* debe ser capaz de reconocer la excepción, almacenar la firma del bloque actual y restaurar dicha firma al final de la rutina de atención a la excepción. El *flag de interrupciones*,

proporcionado a través de la *Interfaz de traza* es usado para reconocer que ha tenido lugar una excepción. Además, el *flag de error* puede ser utilizado para detectar cuando una excepción ha sido causada por un fallo en la ejecución. La detección puede ser mejorada si se proporciona al *Monitor Hardware* información acerca de las excepciones habilitadas en cada sección de código.

Los experimentos han consistido en la inyección de fallos de tipo *SEU* en los registros de la unidad de enteros de manera aleatoria. Los registros de la unidad de enteros se corresponden con el contador de programa y el código de instrucción de cada una de las 7 etapas del pipeline, además de una serie de otros registros especiales y de control. Se han inyectado unos 2000 fallos en registros e instantes de tiempo elegidos aleatoriamente. La aplicación elegida ha sido el algoritmo de ordenación Bubble Sort para un vector de 15 enteros. En este caso no se ha hecho distinción entre errores *SDC* y *HANG* por simplicidad.

Los resultados experimentales obtenidos en esta primera campaña de inyección por simulación se resumen en la Tabla 7. La segunda columna muestra el número de errores detectados por cada una de las técnicas, la tercera columna el porcentaje de errores detectados por cada técnica respecto al total de fallos inyectados, y finalmente la cuarta columna muestra el porcentaje de errores detectados respecto al total de errores.

Los resultados obtenidos muestran que la técnica más efectiva es *Address Checking*, la cual detecta el 62,6 % de los errores, seguida de la técnica de *Monitorización de Firmas*, la cual detecta el 49,3 % de los errores. Ambas técnicas requieren información en tiempo de compilación. Utilizando ambas técnicas a la vez el porcentaje de errores detectados se incrementa en un 7,4 %, detectando un total del 70 % de los errores. Un 7,4 % de los errores es detectado mediante la técnica *Monitorización de Firmas* y no detectado por la técnica *Address Checking*. Este porcentaje de errores afecta al código de instrucción pero no produce un salto ilegal dentro del bloque básico, por tanto la técnica *Address Checking* no tiene capacidad para detectar dichos errores. Sin embargo la técnica de *Monitorización de Firmas*

es capaz de detectar errores que únicamente afecten al código de instrucción, ya que utiliza los mismos para calcular la firma. Si hay algún cambio en el código de instrucción la firma será diferente y por tanto el error será detectado.

	#Fallos	***Total(%)***	***Errores(%)***
Fallos inyectados	1914	100	-
Silencioso	1687	88.1	-
Error	227	11.9	100
Monitorización de firmas	112	5.9	49.3
Address Checking	142	7.4	62.6
Predicción del PC	60	3.1	26.4
Tratamiento excepciones	7	0.4	3.1
Todas las técnicas	166	8.7	73.1

Tabla 7: Campaña de inyección de fallos SEU en simulación para el microprocesador LEON3 con Predicción del PC y Monitorización de Firmas.

La técnica de *Predicción del Contador de Programa* únicamente es capaz de detectar el 26,4 % de los errores, presumiblemente todos aquellos errores que provocan un cambio en el contador de programa. Sin embargo, esta técnica no requiere información en tiempo de compilación y además puede ser fácilmente implementada con un pequeño incremento en el área. En la práctica, todas las técnicas se solapan en gran medida y por tanto se puede concluir que implementar las tres técnicas no aporta una mejora sustancial. Entre Address Checking y Predicción del PC, esta última es preferible porque no requiere información adicional.

Las excepciones provocadas por la inserción el fallo han sido detectadas mediante la observación del *flag de interrupciones* y el *flag de error*. En el caso de los programas de prueba utilizados no hay excepciones esperadas, por tanto, todas las excepciones que se detecten se consideran como error, al ser provocadas por la inyección del fallo.

Utilizando todas las técnicas al mismo tiempo el porcentaje de errores detectados es del 73,1 %. El 26,9 % de errores restantes son principalmente errores de datos, los cuales no pueden ser detectados por las técnicas propuestas, que únicamente son capaces de detectar errores en el flujo de programa.

4.4.2. Descripción del Monitor Hardware

Para los experimentos más extensivos se han implementado las técnicas de *Monitorización de firmas* y *Predicción del Contador de Programa* en el módulo *Monitor Hardware,* tal y como se muestra en la Tabla 8 de manera más detallada.

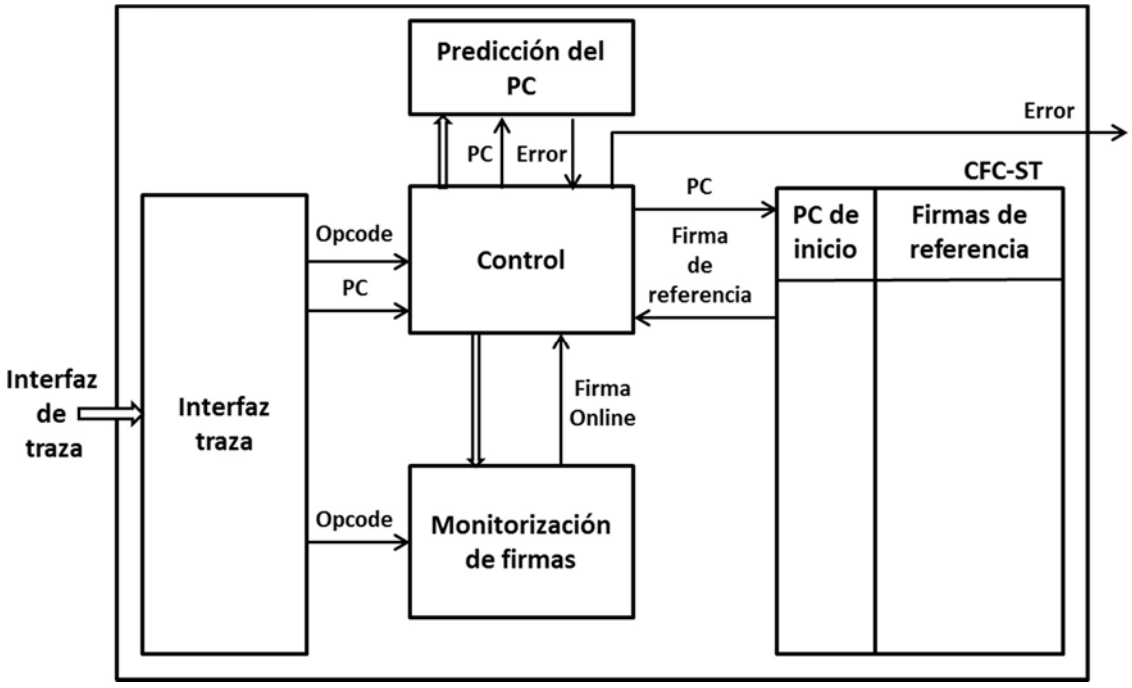

Tabla 8: Esquema del HM (Predicción del PC y Monitorización de firmas) utilizado en los experimentos de simulación en LEON3.

Los distintos bloques de los que se compone el *Monitor Hardware* se describen a continuación:

- *Interfaz de traza*: el bloque *Interfaz de traza* obtiene la información proporcionada a través de la Interfaz de traza, la decodifica y envía la información decodificada al bloque Control y al bloque de *Monitorización de firmas*. Al bloque Control le envía tanto el contador

de programa (*PC*) como el código de instrucción (*opcode*) mientras que al bloque *Monitorización de firmas* únicamente le envía el código de instrucción.

- *Control*: bloque encargado de coordinar el resto de bloques del *Monitor Hardware* y de asegurar el correcto funcionamiento del mismo. El bloque *Control* realiza las siguientes funciones:
    - Indica el final de un bloque básico y por tanto el inicio del siguiente al bloque Monitorización de Firmas. En este momento adquiere la firma online para poder compararla con la de referencia y reinicia el cálculo de la firma del nuevo bloque.
    - Adquiere la firma de referencia de la tabla *CFC-ST* teniendo en cuenta la dirección de inicio del bloque básico que se está ejecutando.
    - Compara la firma online con la firma de referencia indicando con una señal de error cuando difieren.
    - Proporciona al bloque *Predicción del PC* el valor del contador de programa e información decodificada de la instrucción que acaba de ser ejecutada. Indica si la instrucción es de salto, el tipo de instrucción de salto, el desplazamiento del salto, etc..
    - Activa una señal de error cuando el bloque Predicción del PC indica que ha habido una variación en el flujo de programa.
- *Predicción del PC*: bloque encargado de implementar la técnica de *Predicción del contador de programa*. Predice el contador de programa a partir del valor anterior del mismo y de la información decodificada de la última instrucción ejecutada, que es proporcionada por el bloque Control.
- *Monitorización de firmas*: bloque que calcula para cada bloque básico la firma utilizando los códigos de instrucción de cada una de las instrucciones ejecutadas. Una vez que se alcanza el final de un bloque básico, envía la firma calculada al bloque Control y reinicia el valor de la firma.

- Tabla *CFC-ST*: Memoria que almacena las firmas de referencia, calculadas en tiempo de compilación, y las direcciones de inicio de los diferentes bloques básicos.

4.4.3. Características de los experimentos

Se han seleccionado cinco aplicaciones software para la realización de los experimentos:

- *Bubble Sort (BBS):* implementa el algoritmo de ordenación *Bubble Sort* en un vector compuesto por 8 elementos enteros.
- *Multiplicación de matrices (Mmul*t): realiza la multiplicación de dos matrices de tamaño 3x3.
- *Dijkstra:* implementa el algoritmo para la determinación del camino más corto *Dijkstra*, para un grafo de nueve vértices.
- *RLE (Run Length Encoding and Decoding):* implementa el algoritmo de codificación y decodificación *RLE* para un conjunto de cien datos enteros.
- *MF*: implementa el algoritmo de *Ford-Fulkerson*, el cual calcula el flujo máximo en una red de flujo de 32 nodos con al menos 64 arcos.

En la Tabla 9 se detallan las principales características de las aplicaciones utilizadas, incluyendo el tamaño (número de instrucciones), la duración de la ejecución en ciclos de reloj y el número de bloques básicos. En comparación con otros experimentos realizados en esta tesis, se ha limitado el tamaño de los datos para reducir el esfuerzo computacional de la simulación.

	Tamaño (instrucciones)	Duración (ciclos)	Bloques básicos
BBS	26	867	19
MMult	61	1.500	20
Dijkstra	147	2.888	24
RLE	257	7.699	43
MF	744	401.332	172

Tabla 9: Características de los programas utilizados

Se han utilizado tres modelos de inyección de fallos, propuestos originalmente en [NOh02a], y ampliamente utilizados en otros trabajos basados en la detección de errores que afectan al flujo de programa, como los propuestos en [Vemu06][Vemu07]:

- Modelo de inyección de fallos 1: Se cambia de manera aleatoria una instrucción de salto por una instrucción *NOP*. Una instrucción NOP es una instrucción que no produce ningún cambio sobre el estado del programa, salvo que incrementa el contador de programa. Por lo tanto, el contador de programa no se actualizaría con el valor del salto sino que se incrementaría apuntando a la siguiente instrucción.
- Modelo de inyección de fallos 2: se cambia el valor de un bit aleatorio del contador de programa en un ciclo de reloj aleatorio de la ejecución.
- Modelo de inyección de fallos 3: se cambia de manera aleatoria el valor de un bit del operador de una instrucción de salto en un ciclo de reloj aleatorio de la ejecución.

La inyección de fallos se realizó mediante simulación utilizando la herramienta *Modelsim*, siguiendo cada uno de los modelos de fallos descritos. Para cada aplicación software y cada modelo de fallos se han inyectado 10000 fallos. Se ha observado el comportamiento del sistema para cada uno de los fallos inyectados y se ha realizado una clasificación de los mismos teniendo en cuenta el efecto que producen. Los fallos se han clasificado como silenciosos si no producen ningún error en el flujo de programa y como error cuando se ha producido un error en el flujo. Al igual que en los anteriores experimentos de simulación, no se ha hecho distinción entre errores SDC y HANG por simplicidad.

4.4.4. Resultados experimentales

Los resultados experimentales obtenidos en la campaña de inyección de fallos se muestran en la Tabla 10. Estos resultados se han obtenido para una tabla *CFC-ST* lo suficientemente grande como para almacenar todas las firmas y direcciones de inicio de los bloques básicos presentes en cada una de las aplicaciones software ejecutadas. Tanto las firmas como las

direcciones de inicio son almacenadas en la tabla *CFC-ST* antes de iniciar la ejecución con los valores calculados en tiempo de compilación. Los resultados proporcionados en la Tabla 10 muestran que el *Monitor Hardware* es capaz de detectar todos los errores que afectan al flujo de programa sin hacer distinción entre modelos de fallos y aplicaciones software. Esta elevada capacidad para detectar errores puede verse reducida al disminuir el tamaño de la tabla *CFC-ST.* Este resultado muestra una elevada eficiencia en la detección de errores que afectan al flujo de programa para las técnicas *Monitorización de firmas* y *Predicción del contador de programa* trabajando en conjunto, y corrobora los buenos resultados obtenidos en los experimentos preliminares de simulación mostrados en el apartado 4.4.1. En cuanto a la latencia, teniendo en cuenta todos los fallos detectados por ambos métodos, en el peor de los casos tendrá un valor igual al mayor de los bloques básicos.

Una vez realizados los experimentos con una tabla *CFC-ST* capaz de almacenar toda la información necesaria, se ha evaluado el efecto de tener una tabla *CFC-ST* con un tamaño menor que el número de bloques básicos de la aplicación considerada. Al no poder almacenar todas las firmas habrá un conjunto de bloques básicos en los que no se podrá aplicar la técnica de *Monitorización de firmas*, y que por tanto si un error afecta a uno de esos bloques podrá no ser detectado por el módulo *Monitor Hardware*. Por tanto, es necesario seleccionar aquellos bloques básicos más relevantes en la ejecución tratando de reducir el número de errores de flujo no detectados. Se han utilizado las tres técnicas mencionadas en el apartado 3.3.5 para la selección de los bloques más relevantes, que se resumen a continuación:

- Técnica 1: los bloques básicos son ordenados de acuerdo con su tamaño (en término de instrucciones). Se almacenarán en la tabla *CFC-ST* los bloques que ocupen las primeras posiciones en la ordenación. La razón fundamental de esta técnica es que la posibilidad de que un fallo afecte a un bloque básico es proporcional al tamaño del mismo.

- Técnica 2: los bloques básicos son ordenados de acuerdo con el número de veces que son ejecutados durante el transcurso del programa.
- Técnica 3: los bloques básicos se ordenan de acuerdo con el producto del número de veces que son ejecutados y el número de instrucciones que componen dichos bloques. Esta técnica es una combinación de las dos anteriores.

Software	Modelo de inyección de fallos	Silenciosos	Errores	Detectados
BBS	Modelo1	6878	3122	3122
	Modelo2	4757	5243	5243
	Modelo3	9377	623	623
Mmult	Modelo1	4768	5232	5232
	Modelo2	4958	5042	5042
	Modelo3	5423	4577	4577
Dijkstra	Modelo1	6317	3683	3683
	Modelo2	3411	6589	6589
	Modelo3	7890	2110	2110
RLE	Modelo1	7654	2346	2346
	Modelo2	3208	6792	6792
	Modelo3	8827	1173	1173
MF	Modelo1	4496	5504	5504
	Modelo2	3386	6614	6614
	Modelo3	6835	3165	3165

Tabla 10: Campaña de inyección de fallos en simulación para el sistema basado en el microprocesador LEON3 y HM – Predicción del PC y Monitorización de firmas.

En la Tabla 11 se muestran los resultados obtenidos utilizando las tres técnicas de selección de bloques básicos para diferentes tamaños de la tabla *CFC-ST*. Los resultados obtenidos demuestran que con un número de entradas de la tabla *CFC-ST* significativamente inferior al número de bloques básicos las técnicas propuestas pueden alcanzar una cobertura de fallos razonable, corroborando los resultados obtenidos en [Huan99] donde se

trata un problema similar y se hacen un mayor número de experimentos con unas aplicaciones software de mayor tamaño. También se demuestra que la técnica 3 para la selección de bloques básicos es la que proporciona los mejores resultados, siendo mejor que las otras dos técnicas en 21 de 24 casos. Los resultados muestran que la cobertura de fallos, aunque claramente desciende al disminuir el tamaño de la tabla *CFC-ST*, no colapsa, sino que disminuye lentamente, permitiendo buscar una solución de compromiso entre el coste del hardware que supone implementar la tabla *CFC-ST* y la cobertura de fallos. Claramente la eficacia de la técnica de *Monitorización de firmas* se debilita cuando el tamaño del programa supera un límite dado, como sucede en la mayor de las aplicaciones software utilizada en los experimentos.

Finalmente, en la Tabla 12 se muestran los resultados utilizando una tabla dinámica para el almacenamiento de las firmas. En este caso la tabla está inicialmente vacía y se va rellenando cada vez que se entra en un bloque básico. Si no quedan posiciones libres, se reemplaza el bloque menos recientemente usado.

Como se puede observar en la Tabla 12, el método dinámico puede alcanzar una elevada cobertura de fallos, incluso aunque el tamaño de la CFC-ST es bastante más pequeño que el número de bloques básicos. La cobertura de fallos parece ser algo menor en los ejemplos pequeños que en los grandes. Esto se debe a que en el arranque la CFC-ST está vacía y por tanto no puede detectar errores hasta que la tabla se rellena, lo que puede dar lugar a que algunos errores en las fases iniciales de la ejecución no sean detectados. Una vez que la tabla CFC-ST está rellenada, el método dinámico es más efectivo que el estático. Para aplicaciones de software grandes, ésta desventaja inicial es despreciable por lo que el método dinámico resulta más ventajoso.

Aplicación Software (Bloques básicos)	*Tamaño tabla CFC-ST*	*Modelo de inyección de fallos*	*Errores detectados (%)*		
			Técnica 1	*Técnica 2*	*Técnica 3*
Dijkstra (24)	16	Modelo1	97.80	94.46	99.24
		Modelo2	99.94	99.94	99.94
		Modelo3	98.48	98.53	98.53
	8	Modelo1	43.36	77.36	71.63
		Modelo2	99.94	99.94	99.94
		Modelo3	83.84	95.36	91.85
RLE (43)	32	Modelo1	71.44	99.66	99.66
		Modelo2	100	100	100
		Modelo3	90.20	99.06	99.06
	16	Modelo1	46.80	58.65	62.66
		Modelo2	100	100	100
		Modelo3	83.80	91.82	88.92
MF (253)	128	Modelo1	97.52	99.80	99.83
		Modelo2	99.93	99.98	99.98
		Modelo3	87.48	99.80	99.98
	64	Modelo1	87.97	98.77	98.92
		Modelo2	99.93	99.98	99.98
		Modelo3	80.36	99.12	99.32
	32	Modelo1	43.42	98.23	98.23
		Modelo2	99.90	99.98	99.98
		Modelo3	67.58	98.83	98.91
	16	Modelo1	42.85	93.59	93.87
		Modelo2	99.90	99.98	99.98
		Modelo3	67.22	95.08	95.24

Tabla 11: Cobertura a fallos con una tabla CFC-ST de tamaño limitado.

Aplicación Software (Bloques básicos)	*Tamaño tabla CFC-ST*	*Modelo de inyección de fallos*	*Errores detectados (%)*
Dijkstra (24)	16	Modelo1	93,76
		Modelo2	99,94
		Modelo3	95,36
RLE (43)	32	Modelo1	97,53
		Modelo2	100
		Modelo3	99,66
	16	Modelo1	89,6
		Modelo2	100
		Modelo3	95,82
MF (253)	128	Modelo1	99,92
		Modelo2	99,98
		Modelo3	99,96
	64	Modelo1	99,49
		Modelo2	99,98
		Modelo3	99,76
	32	Modelo1	98,97
		Modelo2	99,98
		Modelo3	99,39
	16	Modelo1	97,89
		Modelo2	99,98
		Modelo3	98,14

Tabla 12: Cobertura a fallos con una tabla CFC-ST dinámica.

4.5. Análisis de la Monitorización Dual en el microprocesador LEON3

Para la evaluación de la técnica de Monitorización Dual se utilizó el Monitor Hardware descrito en el apartado 3.4.2 y se implementó un sistema básico con el microprocesador LEON3. Las campañas de inyección han consistido en un conjunto de experimentos de emulación en FPGA utilizando la herramienta AMUSE.

Los resultados experimentales alcanzados en esta campaña de inyección han sido publicados en [Parr14b].

4.5.1. Características de los experimentos

Para la evaluación de la técnica en el microprocesador LEON3 se han utilizado tres programas distintos: un algoritmo de ordenación (BBS), una multiplicación de matrices (Mmult) y un algoritmo de encriptación (AES). El algoritmo de ordenación implementa el método de la burbuja (*Bubble Sort*) con un vector de 15 elementos. El algoritmo Mmult implementa una multiplicación de matrices de 5×5 elementos. Este algoritmo en intensivo en el uso de datos con bucles muy cortos, pasando la mayor parte del tiempo realizando operaciones aritméticas y accediendo a la memoria para obtener datos y almacenar resultados. El algoritmo AES es intensivo en operaciones, principalmente lógicas, pero realiza menos iteraciones y cada iteración es más larga.

Los tres programas escriben resultados parciales en un puerto de salida para facilitar la comprobación de su funcionamiento. Todos los programas fueron escritos en lenguaje C y compilados con GCC usando la opción de optimización –O2. Para evaluar las capacidades de la solución propuesta, los experimentos de inyección de fallos se realizaron con la versión original del software tal como la genera el compilador sin ninguna manipulación. En el capítulo siguiente se muestran además resultados en los que se ha combinado la Monitorización Dual con versiones software endurecidas para errores de datos.

Durante la campaña de inyección se han llevado a cabo experimentos de emulación en *FPGA* inyectando tanto fallos de tipo *SEU* como fallos de tipo *SET*. Respecto a los experimentos de *SEUs,* se ha inyectado un *SEU* en cada registro del microprocesador para cada ciclo de reloj de la ejecución, cubriendo todos los posibles casos de inyección. En los experimentos de *SETs*, se han inyectado fallos en diferentes instantes de tiempo aleatorios dentro de cada ciclo de reloj de la ejecución para todas las puertas que forman el microprocesador. Se ha seleccionado un ancho de pulso del 10% del ciclo de reloj, usando el enfoque descrito en [Entr09].

Los errores han sido clasificados en diferentes categorías, siguiendo la terminología propuesta en [Mukh03]. Los errores que no son detectados por el *Monitor Hardware* son clasificados como *SDC* o como *HANG*. Para determinar cuándo se ha producido un error tipo *HANG* se ha establecido un número de ciclos máximo que el microprocesador puede estar sin enviar datos por el puerto de salida. Un error es clasificado como *HANG* cuando se supera esta condición de número de ciclos máximo sin enviar datos por el puerto.

4.5.2. Resultados experimentales

Se ha realizado la síntesis del sistema basado en el microprocesador *LEON3* y el *Monitor Hardware* con una configuración básica del microprocesador, con las siguientes características:

- 8 ventanas de registros.
- Caché de instrucciones y de datos, de 2 Kbytes cada una.
- Controlador de interrupciones.
- Controlador de memoria
- Interfaz de traza.
- Un puerto de entrada y salida de propósito general.

	#LUTs	***#Registros***
Microprocesador	7.185	1.851
Monitor Hardware	1.230	399
Incremento de área	17,12%	21,55%

Tabla 13: Resultados de síntesis para el diseño basado en el microprocesador LEON3 y HM-Predicción del PC y Monitorización Dual

Los resultados de la síntesis, que se pueden observar en la Tabla 13, muestran como el Monitor Hardware supone un incremento cercano al 22% respecto a un diseño del microprocesador *LEON3* con una configuración mínima y sin tener en cuenta las memorias del banco de registro. Es importante resaltar que el diseño del microprocesador es mínimo y que al utilizar un diseño del *LEON3* más completo, el área del *Monitor Hardware* se mantiene y por tanto el incremento relativa es menor.

Para determinar la eficacia del HM respecto a los errores de control de flujo, los registros internos del procesador se han dividido en dos grupos:

- Grupo I: PC e IR de todas las etapas (346 biestables), De acuerdo con [Vemu07], los fallos en estos registros determinan los errores de control de flujo.
- Grupo II: resto de los registros (1505 biestables).

Los resultados se muestran en las Tablas 14, 15 y 16 para las tres aplicaciones software, respectivamente. Las tres primeras filas de cada tabla muestran los resultados de la inyección de fallos *SEUs* en el grupo I, en el grupo II y en la totalidad de registros respectivamente. La última fila muestra los resultados obtenidos en la inyección de fallos *SETs.* Estos últimos no se han segregado en grupos puesto que no se puede asociar con precisión cada puerta lógica a funciones de control de flujo. De izquierda a derecha, las tablas muestran el total de fallos inyectados, los errores observados, el número de errores *SDC*, el número de errores *HANG* y los errores detectados por el *Monitor Hardware*.

Elementos	*Fallos inyectados*	*Errores observados*	*SDC*	*Hang*	*Errores detectados*
PC & IR (I)	1,177 M	343.278	0	0	343.278 (100%)
Otros Regs. (II)	5,123 M	361.307	167.192 (46,3%)	36.764 (10,2%)	157.351 (43,6%)
Regs. (todos)	6,301 M	704.585	167.192 (23,7%)	36.764 (5,2%)	500.629 (71,1%)
Lógica comb. (SETs)	80,649 M	777.634	258.768 (33,3%)	24.579 (3,2%)	494.287 (63,6%)

Tabla 14: Resultados de inyección de fallos con Monitorización Dual, (Bubble Sort)

Elementos	*Fallos inyectados*	*Errores observados*	*SDC*	*Hang*	*Errores detectados*
PC & IR (I)	2,125 M	466.416	0	0	466.416 (100%)
Otros Regs. (II)	9,245 M	599.949	264.051 (44,0%)	50.084 (8,3%)	285.814 (47,6%)
Regs. (todos)	11,371 M	1.066.365	264.051 (24,8%)	50.084 (4,7%)	752.230 (70,5%)
Lógica comb. (SETs)	145,533 M	1.495.468	436.119 (29,2%)	25.499 (1,7%)	1.033.850 (69,1%)

Tabla 15: Resultados de inyección de fallos con Monitorización Dual. (Mmult)

Elementos	*Fallos inyectados*	*Errores observados*	*SDC*	*Hang*	*Errores detectados*
PC & IR (I)	1,514 M	833.840	0	0	833.840 (100%)
Otros Regs. (II)	6,587M	610.326	321.583 (52,7%)	11.968 (2,0%)	276.775 (45,31%)
Regs. (todos)	8,102 M	1.444.166	321.583 (22,3%)	11.968 (0,8%)	1.110.615 (76,9%)
Lógica comb. (SETs)	103,701 M	1.017.164	324.243 (31,9%)	1.439 (0,1%)	691.482 (68,0%)

Tabla 16: Resultados de inyección de fallos con Monitorización Dual, (AES)

Los resultados obtenidos utilizando la aplicación software *Bubble Sort* se muestran en la Tabla 14. El algoritmo *Bubble Sort t*arda 3.404 ciclos de reloj en ordenar un vector de 15 posiciones, por tanto, se han inyectado 3.404 *SEUs* por registro, alcanzando un total de 6,3 millones de fallos inyectados. También se han inyectado 10.234 *SETs* por nodo combinacional, alcanzando un total de 80,6 millones de *SETs*. El *Monitor Hardware* es capaz de detectar la totalidad de errores en el grupo de registros I, el formado por el contador de programa y el código de operación de todas las etapas del pipeline, y muchos de los errores que forman parte del grupo de registros II. Aunque el grupo de registros I es mucho más pequeño que el grupo de registros II, acumula alrededor de la mitad de los errores observados. Esto es debido a que el contador de programa y el registro de instrucción son críticos para el correcto funcionamiento del microprocesador. En particular, los errores del grupo de registros I son en su totalidad errores que afectan al flujo de programa [Vemu07], mientras que los errores que forman parte del grupo de registros II pueden producir una amplia variedad de efectos, pudiendo ser detectados por el *Monitor Hardware* si el error termina provocando un error en el flujo de programa, como por ejemplo, un bucle infinito, una dirección

inválida, etc. Si se tienen en cuenta todos los registros, el *Monitor Hardware* es capaz de detectar en torno al 71% de los errores *SEU* observados. Este resultado está en consonancia con los que se obtuvieron en los experimentos preliminares presentados en el apartado 4.4.1. Por otra parte, el *Monitor Hardware* es capaz de detectar un porcentaje similar de fallos *SETs,* alcanzando una detección del 63%.

Los resultados de los experimentos de inyección utilizando la *Multiplicación de matrices* como caso de estudio se muestran en la Tabla 15. La Multiplicación de matrices requiere 6.143 ciclos de reloj para completar la multiplicación de dos matrices de 5x5. En este caso se inyectaron un total de 11,3 millones de *SEUs* y 145,5 millones de *SETs* para alcanzar la misma cobertura de inyección de fallos que en los experimentos realizados con el algoritmo *Bubble Sort*. Los resultados que se obtienen son muy similares a los obtenidos con el algoritmo *Bubble Sort*. El *Monitor Hardware* es capaz de detectar todos los errores del grupo de registros I y muchos de los errores del grupo de registros II, detectando el 70% de los errores *SEUs*. En cuanto a los experimentos de inyección de fallos *SET* se consigue alcanzar una detección del 69%.

Finalmente, los resultados utilizando la aplicación AES se muestran en la Tabla 16. Esta aplicación tiene un código más grande, aunque se ejecuta en menos ciclos de reloj, concretamente 4377 ciclos. Los resultados son de nuevo similares a los de las otras aplicaciones.

Observando los resultados obtenidos se puede deducir que el *Monitor Hardware* es capaz de detectar la totalidad de los errores que afectan al flujo de programa, ya que es capaz de detectar todos los errores en el contador de programa y en el código de instrucción. Además se detectan unos porcentajes importantes de errores en otros registros, que acaban afectando al flujo de programa. Con la finalidad de corroborar los resultados obtenidos en esta campaña de inyección fallos se ha realizado un segunda campaña de inyección de fallos tanto *SEUs* como *SETs* en los que se ha combinado con un endurecimiento software. Estos experimentos son descritos en el capítulo 5.

4.6. Conclusiones

El objetivo de los diferentes experimentos descritos en este capítulo es comprobar la capacidad para detectar errores que afectan al flujo de programa de cada una de las técnicas de *CFC* presentadas en esta tesis.

Se han llevado a cabo campañas de inyección de fallos realizando tanto experimentos de simulación como experimentos de emulación en FPGA. Además se han utilizado dos microprocesadores diferentes, el LEON3 y el Picoblaze. Los primeros experimentos se han realizado en el microprocesador Picoblaze con el objetivo de validar la técnica de Predicción del Contador de Programa en un microprocesador sencillo. Posteriormente se han llevado a cabo experimentos más amplios en el microprocesador LEON3 para la Monitorización de Firmas y la Monitorización Dual.

Para la técnica de Monitorización de Firmas se ha utilizado la inyección de fallos mediante simulación con la herramienta Modelsim, debido a la necesidad de utilizar los modelos de fallos propuestos por otros autores. El número de fallos que se puede inyectar mediante simulación es relativamente pequeño. No obstante, se inyectaron 10000 fallos para cada caso de prueba, aunque para ello fue necesario un esfuerzo computacional muy elevado que se obtuvo repartiendo la tarea entre varios computadores. Para la técnica de Monitorización Dual se utilizó la emulación en FPGA con la herramienta AMUSE. Gracias a AMUSE se han realizado campañas exhaustivas para SEUs, considerando todos los fallos en todos los registros y en todos los ciclos de reloj. Asimismo, para los fallos de tipo SET se inyectaron también un número amplio de fallos en todas las puertas lógicas y en todos los ciclos de reloj. Es importante destacar que AMUSE es el sistema de inyección de fallos más potente que existe en la actualidad y que en la literatura no se encuentran técnicas que hayan sido validadas con campañas de inyección de fallos tan grandes. Gracias a estas capacidades de AMUSE ha sido posible demostrar fehacientemente las ventajas de la técnica de Monitorización Dual.

Los resultados obtenidos en los experimentos demuestran que las técnicas de *CFC* propuestas en este trabajo de tesis son capaces de detectar un porcentaje muy elevado de los errores que afectan al flujo de programa, alcanzando una cobertura total cuando se utilizan varias técnicas conjuntamente.

La técnica de *Predicción del contador de programa* es capaz de detectar un subconjunto de errores, concretamente los que afectan al contador de programa. Sin embargo, es muy eficiente para detectar este tipo de errores y además puede ser implementada con un conjunto de recursos muy reducido, ya que no precisa almacenar información en tiempo de compilación. Para el caso de un microprocesador sencillo como Picoblaze, esta técnica es capaz de detectar la mayoría de los errores de tipo HANG. Por otra parte, esta técnica es complementaria a las otras dos técnicas descritas en la tesis, *Monitorización de firmas* y *Monitorización Dual*, y necesaria para que éstas alcancen elevadas tasas de detección de errores de control de flujo, como se puede observar en los experimentos realizados en el microprocesador *LEON3* y descritos en los apartados 4.4 y 4.5, en los que se alcanza el 100% de detección en errores en el flujo de programa.

La técnica de *Monitorización de Firmas* es capaz de detectar todos aquellos errores que afectan al código de instrucción siempre que se almacenen todas las firmas de referencia calculadas en tiempo de compilación, bajando levemente la capacidad de detección si la tabla *CFC-ST* no puede almacenar todas las firmas. Trabajando en conjunto con la técnica *Predicción del contador de programa* alcanza un 100% de detección de errores que afectan al flujo de programa, tal y como se muestra en el apartado 4.4. Para aplicaciones de software grandes, se ha propuesto también una solución dinámica que puede trabajar con una tabla CFC-ST de tamaño limitado, optimizando la capacidad de detección de errores. El método dinámico produce resultados próximos a la detección completa y en general es más efectivo que el método estático.

Por último, la técnica de *Monitorización Dual* presenta una elevada eficiencia para detectar errores en el flujo de programa y además puede ser implementada sin necesidad de almacenar información de referencia, lo que supone que el impacto sobre el área del sistema es muy bajo. Esta técnica detecta errores que afecten al contador de programa o al código de instrucción en cualquiera de las etapas del pipeline, comparando la información obtenida desde dos puntos de observación diferentes, la *interfaz de traza* y el bus de la memoria. Combinando esta técnica con *Predicción del contador de programa* se alcanza un 100% de detección de errores que afectan al flujo de programa, tal y como se muestra en el apartado 4.5.

A la vista de los resultados se puede concluir que las técnicas presentadas en esta tesis protegen de una manera eficiente frente a errores en el flujo de programa alcanzando una buena solución de compromiso entre la cobertura a fallos y el impacto sobre el rendimiento del sistema. Además, todas las técnicas pueden implementarse de manera no intrusiva, lo que las hace susceptibles de ser utilizadas con cualquier procesados sin necesidad de modificarlo, siempre que se disponga de una interfaz de traza elemental.

No obstante, para una solución completa es necesario tener en cuenta los errores que afectan a los datos. Aunque las técnicas propuestas solo cubren los errores de control de flujo, cabe destacar que éstas alcanzan porcentajes cercanos al 70% de cobertura cuando se consideran todos los posibles fallos, como se ha demostrado en el apartado 4.5. Para conseguir alcanzar coberturas de fallos cercanas al 100% es preciso considerar técnicas híbridas, las cuales combinan las técnicas de *CFC* con técnicas de endurecimiento software. En el capítulo siguiente se analizan los resultados que se pueden obtener mediante la combinación de las técnicas propuestas en el capítulo 3 y técnicas de endurecimiento para errores de datos implementadas en software.

CAPÍTULO 5

RESULTADOS EXPERIMENTALES UTILIZANDO TÉCNICAS DE ENDURECIMIENTO SOFTWARE

5. Resultados experimentales utilizando técnicas de endurecimiento software

A lo largo de esta tesis se han propuesto y validado diversas técnicas de detección de errores de control de flujo en microprocesadores. Aunque estos errores son generalmente los más relevantes, para alcanzar altas tasas de cobertura de fallos es necesario complementarlas con técnicas de detección de errores de datos. Para comprobar la eficacia de las técnicas propuestas y su dependencia respecto al software utilizado, se han realizado nuevos experimentos utilizando software endurecido para errores de datos. En este capítulo se muestran los resultados de estos experimentos.

Al igual que en las campañas de inyección mostradas en el Capítulo 4, los experimentos se han llevado a cabo en dos microprocesadores diferentes, Picoblaze y LEON3. Sin embargo, en este caso únicamente se han realizado experimentos de emulación en FPGA, utilizando la herramienta AMUSE para la realización de las campañas de inyección. Gracias a AMUSE ha sido posible realizar campañas de inyección de millones de fallos, tanto para SEUs como SETs.

En este capítulo se describen en primer lugar las técnicas de endurecimiento software seleccionadas. A continuación se presentan los resultados obtenidos con el microprocesador Picoblaze combinando la técnica de Predicción del Contador de Programa con software endurecido para errores de datos. Posteriormente se presentan los resultados obtenidos con el microprocesador LEON3, analizando la aportación de diferentes técnicas de endurecimiento software y de la técnica de Monitorización Dual. Finalmente se presentan las conclusiones de este capítulo.

5.1. Introducción

La combinación de las técnicas de detección de errores de control de flujo propuestas en esta tesis, que son técnicas implementadas mediante hardware externo, y las técnicas de endurecimiento para errores de datos, implementadas mediante software, da lugar a soluciones híbridas. Estas

soluciones tratan de aprovechar las ventajas inherentes a cada tipo de implementación, hardware o software, en el dominio en el que son más eficientes. Por un lado, la implementación hardware de técnicas de detección de errores de control de flujo permite una cobertura total de este tipo de errores con un coste muy pequeño en área, de manera no intrusiva y sin penalizar las prestaciones. Sin embargo, la detección de errores de datos mediante hardware externo es difícilmente generalizable, y en todo caso es muy costosa, ya que en la práctica requiere, de alguna forma, de la replicación de las rutas de datos [Berg09]. Por el contrario, una implementación puramente software introduce una gran penalización en prestaciones, particularmente para el caso de la detección de errores de control de flujo [Azam11a], [Azam13]. Además, se ha demostrado que las técnicas de software existentes no permiten detectar todos los errores de control de flujo [Azam11a], [Azam12], [Azam13], debido a que se ven inherentemente afectadas por las propias alteraciones del flujo de programa.

Una solución híbrida que combine la detección de errores de control de flujo mediante hardware externo con la detección de errores de datos mediante software presenta teóricamente las siguientes ventajas:

- No se requiere la modificación del hardware del procesador.
- Proporcionan gran flexibilidad, permitiendo una selección de técnicas hardware o software de acuerdo con las necesidades de cada aplicación y las características del microprocesador utilizado.
- Permite detectar todos los errores de control de flujo, como se ha demostrado en el capítulo anterior
- Reducen la penalización en el tiempo de ejecución.

A la vista de los resultados experimentales obtenidos en las campañas de inyección de fallos mostradas en el capítulo 4, se puede afirmar que las técnicas presentadas en este trabajo de tesis se caracterizan por alcanzar una cobertura a fallos del 100% frente a errores que afectan al flujo de programa y con un impacto sobre el rendimiento del sistema bajo, ya que

pueden ser implementadas sin necesidad de almacenar información de la ejecución calculada en tiempo de compilación. Sin embargo, para alcanzar una protección completa frente a errores es preciso detectar tanto errores en el flujo de programa como errores en los datos.

El principal objetivo de los experimentos que se presentan en este capítulo es evaluar la capacidad total de detección de errores de soluciones híbridas que utilicen las técnicas propuestas en esta tesis para la detección de errores de control de flujo. En particular, se pretende evaluar la cobertura de todos los posibles fallos de tipo SEU y de tipo SET, tanto de flujo como de datos. Para la detección de errores de datos se han seleccionado una serie de técnicas de endurecimiento software que se describen en el apartado siguiente.

5.2. Técnicas de endurecimiento software seleccionadas

Entre las técnicas de endurecimiento software existentes se han seleccionado algunas de las más recientes para realizar las diferentes campañas de inyección de fallos: *SWIFT-R*, *Replicación de procedimientos*, *Data duplication*, *Final variables e Inverted branches.* Para la elección de estas técnicas se han tenido en cuenta diversos factores. En primer lugar, todas ellas son técnicas relativamente recientes, que se han obtenido mediante refinamientos de técnicas anteriores, por lo que en general son técnicas bastante maduras. En segundo lugar, se ha tenido en cuenta la facilidad de implementación. Generalmente, estas técnicas requieren la modificación a nivel de código ensamblador, lo que resulta muy complejo y en la práctica requiere la modificación del compilador.

Las técnicas *SWIFT-R* y *Replicación de procedimientos* han sido utilizadas en la campaña de inyección de fallos en la que se utiliza el microprocesador *Picoblaze* como caso de estudio. Para este propósito se ha colaborado con el Grupo de la Universidad de Alicante liderado por el Profesor S. Cuenca, que ha desarrollado un compilador de endurecimiento para el microprocesador *Picoblaze*, denominado *SHE* (*Software Hardening Environment*). Este compilador soporta las técnicas *SWIFT-R* y Replicación de

procedimientos. Para los los experimentos con el microprocesador *LEON3* se ha utilizado una combinación del resto de las técnicas (*Data duplication*, *Final variables e Inverted branches)*, por su mayor facilidad de implementación. Dado que en este caso no se disponía de un compilador apropiado para generar software endurecido, la implementación se ha realizado directamente en alto nivel (lenguaje C).

5.2.1. SWIFT-R

La técnica *SWIFT-R* [Reis07] es un método global destinado a recuperar la ejecución frente a fallos de datos, utilizando la redundancia de variables y registros a nivel de instrucción. Esta técnica se basa principalmente en mantener copias de cada uno de los datos utilizados en la zona del programa que se quiere proteger, llamada Esfera de Replicación (SoR). Las instrucciones que operan con dichos datos se replican también, de manera que se computan flujos de datos redundantes e independientes. Posteriormente se realizan comprobaciones de consistencia de los datos cuando éstos salen de la zona protegida y también antes de cada salto condicional.

La técnica *SWIFT-R* es una mejora de la técnica *SWIFT* [Reis05]. La técnica *SWIFT* habilita al compilador para que duplique las instrucciones del programa. Tanto los datos duplicados como los originales se almacenan en registros diferentes, de manera que no interfieran entre ellos. Además se introducen ciertos puntos de validación en el programa en los que se comprueba que los datos generados por las instrucciones originales y duplicadas son los mismos. Los puntos de validación son insertados antes de cualquier instrucción que pueda potencialmente generar un dato de salida. Suponiendo que todas las salidas están asociadas a posiciones de memoria (*memory mapped*), un programa se ejecuta correctamente si todas las instrucciones de *store* se ejecutan correctamente con datos correctos. Teniendo en cuenta esta suposición, los puntos de validación se insertan antes de todas las instrucciones de *store*. A diferencia de la técnica *SWIFT,* la técnica *SWIFT-R* realiza dos copias de las instrucciones y los datos en lugar de una. Al tener tres copias, aunque un fallo afecte a una de las copias

seguiremos teniendo dos copias con el valor correcto, lo que permite corregir errores simples mediante un votador.

#	*Código no endurecido*	*Código endurecido con SWIFT- R*
1	`main: LOAD s0, 00`	`main: LOAD s0, 00`
2		*Create* s0 *copies*
3	`LOAD s1, 2A`	`LOAD s1, 2A`
4		*Create* s1 *copies*
5	`ADD s0, s1`	`ADD s0, s1`
6		`ADD s0', s1'`
7		`ADD s0'', s1''`
8	`CALL incr`	`CALL incr`
9		*Majority voter for* s0
10	`STORE s0, 00`	`STORE s0, 00`
11	`RETURN`	`RETURN`
12		
13	`incr: LOAD s2, 0F`	`incr: LOAD s2, 0F`
14		*Create* s2 *copies*
15	`ADD s0, s2`	`ADD s0, s2`
16		`ADD s0', s2'`
17		`ADD s0'', s2''`
18	`RETURN`	`RETURN`

Figura 18 : Ejemplo de código endurecido con la técnica SWIFT-R.

La Figura 18 presenta un ejemplo de un código básico en ensamblador endurecido utilizando la técnica *SWIFT-R*. Como se puede observar, las copias s0', s0'', etc… se almacenan en otros registros que no son utilizados en el programa original. Los votadores son procedimientos de recuperación que comparan si al menos dos versiones de un dato tienen el mismo valor, corrigiendo un posible dato corrupto con la tercera copia. Cualquier error que afecte a los datos de programa es enmascarado por las copias de los datos y corregido en un corto plazo. La corrección de un dato tarda un número de ciclos igual a la suma de los ciclos necesarios para ejecutar las instrucciones del votador más los ciclos necesarios para recuperar el registro afectado por el error.

Como consecuencia de la replicación, el tamaño del código y el tiempo de ejecución pueden ser incrementados entre 2,5 y 3 veces respecto al programa original. Para tratar de disminuir el impacto sobre el sistema se pueden seleccionar zonas de protección en lugar de proteger todo el código. En general, se asume que la memoria está protegida y por tanto no

produce datos incorrectos, lo cual se consigue habitualmente utilizando un código corrector de error. En caso contrario, sería necesario introducir la memoria en la Esfera de Replicación y todos los datos almacenados en ella deben ser replicados.

Para optimizar la robustez del sistema frente al consumo de recursos, se debe buscar una solución de compromiso entre el incremento del código, el tiempo de ejecución y la zona protegida, buscando proteger aquellas zonas del código más sensibles. Este método es ideal cuando se requiere una rápida respuesta frente al error y no hay limitaciones en cuanto al tiempo de ejecución y el tamaño de la memoria del sistema.

5.2.2. Replicación de procedimientos

La técnica de replicación de procedimientos se basa, como su nombre indica, en replicar los procedimientos en vez de replicar las instrucciones. La unidad de replicación es un procedimiento (función). Un procedimiento es un bloque de código que realiza una única tarea y que devuelve algunos valores al finalizar dicha tarea. Cada procedimiento es ejecutado dos veces y re-ejecutado una tercera vez si se produce alguna diferencia entre las dos ejecuciones anteriores [NOh02b]. En la Figura 19 se muestra el funcionamiento de este método.

Para implementar la técnica de *Replicación de procedimientos* se requieren unas pequeñas transformaciones en el código original, relacionadas con los saltos condicionales y con las comprobaciones de consistencia, las cuales se introducen después de las llamadas a los procedimientos. Por lo tanto, el impacto sobre el tamaño del código es a priori pequeño. Sin embargo, la latencia depende del número de instrucciones de las que esté formado el procedimiento, y por tanto, puede ser elevado. Normalmente, el tiempo de recuperación es igual al tiempo de ejecución del procedimiento añadiendo el tiempo necesario para ejecutar unas pocas comparaciones. En una ejecución normal libre de errores el incremento del tiempo de ejecución es de dos veces, mientras que si se ha producido un error se tendrá que ejecutar el procedimiento una tercera vez,

incrementando el tiempo de ejecución hasta tres veces. Por tanto, se tendrán que tener en cuenta tanto el tiempo de recuperación como el incremento del tiempo de ejecución a la hora de seleccionar este método para realizar un endurecimiento software del código a ejecutar.

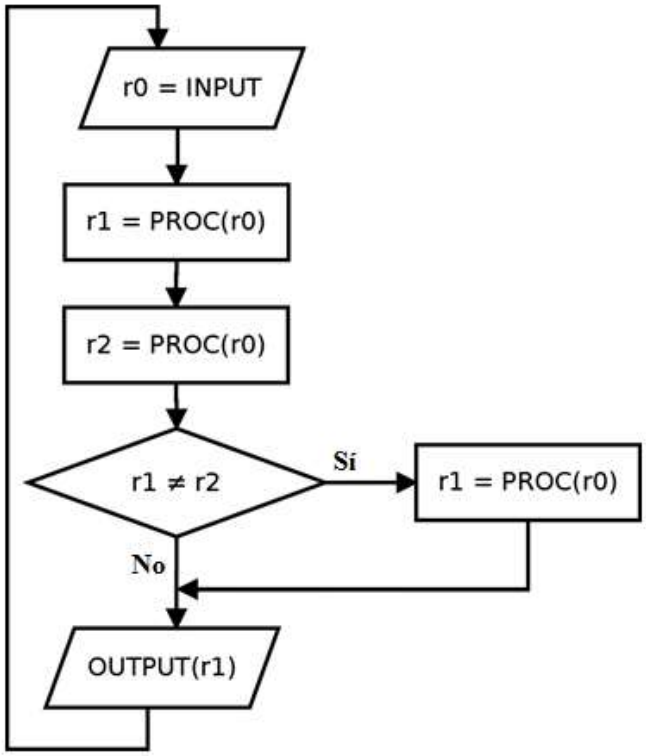

Figura 19: Diagrama de flujo que ilustra la técnica de endurecimiento software utilizando la técnica de *Replicación de procedimientos*.

5.2.3. Data duplication

La técnica *Data Duplication* está basada en [Reba99]. Los autores proponen un conjunto de reglas que pueden ser aplicadas directamente sobre el código a alto nivel. Estas reglas se pueden dividir en dos conjuntos de reglas: las que afectan a los datos y las que afectan al código.

A continuación se muestra el conjunto de reglas que afectan a los datos:

- Regla 1: cada variable del programa debe ser duplicada. Los parámetros de los procedimientos, los valores de retorno de los procedimientos y los índices de los bucles son considerados como variables y por tanto, deben ser duplicados.

- Regla 2: cada una de las operaciones de escritura realizadas sobre las variables deben ser realizadas también en las variables duplicadas.
- Regla 3: después de cada operación de lectura de una variable, las dos copias de dicha variables deben ser comparadas con la finalidad de detectar incongruencias. En caso de que haya alguna diferencia entre ambas variables se activa una señal de error. La comparación entre ambas variables se realiza inmediatamente después de cada operación de lectura con la finalidad de bloquear una posible propagación del error.

Esta técnica puede alcanzar una cobertura a fallos cercana al 100% con una latencia mínima. La latencia es aproximadamente igual a la distancia temporal entre el fallo y la primera lectura de la variable afectada.

Los fallos que afectan al flujo de programa requieren un conjunto de reglas extendidas descritas en [Reba99]. Este conjunto de reglas extendida utiliza bloques básicos para identificar las diferentes partes del código y hacen posible la detección de errores en el flujo de programa.

5.2.4. Final Variables

La técnica *Final variables* está basada en [Nico03]. Se basa principalmente en un conjunto de reglas utilizadas para detectar errores que afectan a los datos del programa:

- Regla 1: En primer lugar se debe de identificar la relación que existe entre las diferentes variables del programa.
- Regla 2: Las variables deben de ser clasificadas de acuerdo con la importancia dentro del programa, se hará distinción entre variables intermedias y variables finales.
- Regla 3: Todas las variables del programa deben ser duplicadas.
- Regla 4: Todas las operaciones que se realicen a lo largo de la ejecución del programa deben ser realizadas en ambas copias de las variables.

- Regla 5: Siempre que se realice una escritura en una variable final las dos copias deben ser comparadas con el fin de buscar posibles diferencias en las dos copias. En caso que las copias de la variable final sean diferentes se habrá detectado un error en los datos.

Según los autores *Final Variables* es capaz de alcanzar una cobertura a fallos del 100% cuando se realiza una campaña de inyección en la que únicamente se inyectan errores en la memoria. Al igual que en la técnica *Data duplication* esta técnica se complementa con un conjunto de reglas que se utilizan para detectar errores en el flujo de programa.

5.2.5. Inverted Branches + Variables

La técnica *Inverted branches-Variables* está basada en [Azam12] donde se propone una solución híbrida. Se utiliza un módulo hardware para detectar errores en el flujo de programa y se combina con dos técnicas de endurecimiento software: *Inverted branches* y *Variables*.

A continuación se describen ambas técnicas:

- *Inverted branches*: esta técnica detecta errores de datos en instrucciones de salto condicional. Cuando se ejecuta una instrucción de salto condicional el flujo de programa puede seguir dos posibles caminos, tomar el salto e incrementar el contador de programa con el desplazamiento del salto, o no tomar el salto e incrementar el contador de programa hasta la siguiente instrucción. Sin embargo, sólo una de las dos opciones es la correcta. Un error que afecte al código que evalúa la condición de salto puede provocar que se ejecute la opción incorrecta. Normalmente este error no puede ser detectado por técnicas de *CFC*, ya que realmente es un error en los datos que acaba afectando al flujo de programa. La técnica *Inverted branches* detecta este tipo de errores replicando las instrucciones de salto y evaluando las condiciones de salto dos veces. Si el salto ha sido tomado se repite la instrucción de salto invirtiendo la condición. Si no se ha tomado el salto, se repite la instrucción de

salto manteniendo la misma condición. Si la instrucción de salto que se repite es tomada, la evaluación de la condición habrá cambiado y se habrá detectado un error. La Figura 20 muestra la metodología seguida para detectar errores en la condición de salto.

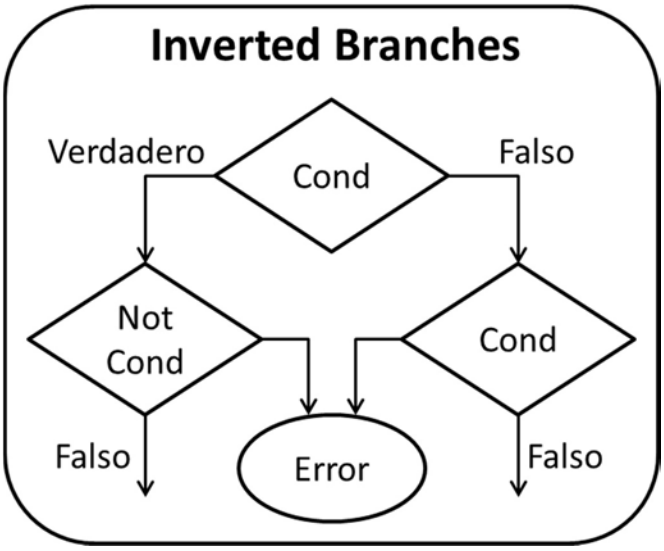

Figura 20: Esquema de la técnica de endurecimiento software *Inverted branches*.

- *Variables*: En esta técnica se duplican una serie de variables del programa. Cada vez que se realiza una operación sobre una variable se actualiza la variable duplicada y se realizan comprobaciones de consistencia cada vez que se realizan accesos a memoria o cuando se ejecuta una instrucción que afecte al flujo de programa. Se debe decidir el número de variables a duplicar dependiendo de los requisitos de la aplicación.

5.3. Análisis con el microprocesador Picoblaze como caso de estudio

En primer lugar se ha realizado una campaña de inyección de fallos utilizando el microprocesador *Picoblaze* como caso de estudio. Esta campaña de inyección de fallos es equivalente a la realizada en el apartado 4.3, excepto porque en este caso se combina la técnica de *Predicción del contador del programa* con varias técnicas de endurecimiento software implementando, por tanto, una técnica híbrida.

Con estos experimentos se pretende evaluar la capacidad de detección de errores de la técnica de *Predicción del contador de programa* al trabajar conjuntamente con técnicas de endurecimiento software. Los resultados experimentales obtenidos en esta campaña de inyección han sido publicados en [Parr14a].

5.3.1. Características de los experimentos

La campaña de inyección de fallos ha consistido en una serie de experimentos de emulación en *FPGA* en los que se han inyectado fallos *SET* y *SEU* en un sistema basado en el microprocesador *Picoblaze*. Los experimentos han sido realizados utilizando la herramienta *AMUSE* [Entr12*]*, la cual ha permitido cubrir los fallos de tipo SEU de manera exhaustiva y los fallos de tipo SET con una muestra muy amplia.

En todos los aspectos, los experimentos se han realizado de manera similar a los descritos en el apartado 4.3 para las versiones no endurecidas del software. Se han utilizado las mismas aplicaciones software (*Mmult, PID y FIR*), las cuales se han endurecido utilizando la herramienta SHE para las dos técnicas seleccionadas, *SWIFT-R* y Replicación de procedimientos. Como se mencionó anteriormente, se han seleccionado estas dos técnicas porque adoptan distintas estrategias y obtienen distintos resultados en cuanto al incremento de área, incremento del tiempo de ejecución y latencia de detección, y por tanto, pueden aplicarse en distintos escenarios dependiendo de los requerimientos del sistema. Además, la implementación de estas técnicas está soportada por las herramientas desarrolladas por la Universidad de Alicante. Por otra parte, el *Monitor Hardware* utilizado es el mismo, puesto que es independiente de la versión software que se utilice sin que se requiera ninguna modificación adicional.

A la hora de seleccionar una técnica de endurecimiento software, dos de los factores más relevantes son el impacto que produce en el tamaño de código, y en consecuencia en los requerimientos de memoria para almacenarlo, y en el tiempo de ejecución. En la Tabla 12 se resumen las características de las aplicaciones software utilizadas en los experimentos en

atención a estos parámetros. En la segunda columna se muestra el tamaño del código original. En la tercera y cuarta columnas se muestran los factores de aumento (en número de veces) del tamaño de código y del tiempo de ejecución, respectivamente. La técnica *SWIFT-R* es ligeramente peor que la técnica *Replicación de procedimientos*, lo que se debe a que en la primera se introducen comprobaciones más frecuentes. Sin embargo, la técnica *Replicación de procedimientos* presenta una latencia de detección elevada, mientras que en la técnica *SWIFT-R* la latencia es despreciable. Dependiendo de los requisitos de cada aplicación se puede seleccionar una técnica u otra.

Programa	**Instrucciones (tamaño)**	**Versión**	**Incremento de tamaño de código**	**Incremento tiempo de ejecución**
Mmult	3304	SWIFT-R	x3,02	x2,55
		Replicación de procedimientos	x1,65	x2,03
PID	7390	SWIFT-R	x2,37	x2,65
		Replicación de procedimientos	x1,24	x2,02
FIR	7374	SWIFT-R	x2,85	x2,68
		Replicación de procedimientos	x1,07	x1,96

Tabla 17: Incrementos de área y tiempos de ejecución para las técnicas SWIFT-R y Replicación de procedimientos.

Como se ha indicado anteriormente, en los experimentos se han inyectado tanto fallos de tipo *SEU* como de tipo *SET*. En los experimentos de inyección de fallos de tipo *SEU* se han inyectado fallos en todos los registros del microprocesador en cada uno de los ciclos de reloj del programa. En los experimentos de inyección de fallos de tipo *SET* se han inyectado fallos en todas las puertas del microprocesador en instantes aleatorios del ciclo de reloj, con un ancho de pulso del fallo de aproximadamente el 20% del periodo del ciclo de reloj, que para el circuito evaluado es de 500 ps. La

clasificación de fallos se ha realizado también de la misma manera que en los experimentos reportados en el apartado 4.3, utilizando las categorías de fallos silenciosos, SDC, HANG y detectados. Esta última categoría incluye los fallos detectados tanto por el Monitor Hardware como por el software endurecido.

5.3.2. Resultados experimentales

Los resultados obtenidos en la campaña de inyección de fallos *SEU* y *SET* en el sistema basado en el microprocesador *Picoblaze* se muestran en la Tabla 18 y en la Tabla 19 respectivamente. Ambas tablas proporcionan los resultados obtenidos implementando las dos técnicas de endurecimiento software sin incluir el *Monitor Hardware: SWIFT-R* (*SR*) y *Replicación de procedimientos* (*RP*), así como los resultados obtenidos implementando ambas técnicas trabajando en conjunto con el *Monitor Hardware* (*SR + HM*, *RP+HM*).

Los resultados experimentales demuestran que gracias a la combinación de las técnicas de endurecimiento software, capaces de detectar errores en los datos, y el *Monitor Hardware*, capaz de detectar errores en el flujo de programa, se pueden mitigar casi todos los errores que se producen durante la ejecución de cualquiera de las aplicaciones software utilizadas como caso de estudio. Generalmente el *Monitor Hardware* detecta la mayoría de los errores *HANG* y las técnicas de endurecimiento software detectan los errores *SDC*, pero se requiere que trabajen conjuntamente para alcanzar una mitigación relevante.

Las técnicas de endurecimiento software utilizadas no tienen la capacidad de detectar errores de tipo *HANG*. Sin embargo, la técnica de *Predicción del contador de programa* es capaz de detectar los errores que afectan al contador de programa. Por otro lado, los errores tipo *SDC* son generalmente errores en los datos que no afectan al flujo de programa y, por tanto no pueden ser detectados en su mayoría por el *Monitor Hardware,* siendo necesario la utilización de técnicas de endurecimiento software para detectarlos.

La última columna de ambas tablas muestra la reducción de la tasa de error relativa, calculada respecto a la versión no endurecida. En el mejor de los casos (*FIR*) se consigue reducir 114 veces la tasa de error relativa, lo que representa una mejora de 2 órdenes de magnitud.

Las dos técnicas de endurecimiento software utilizadas tienen diferentes tiempos de recuperación y producen también diferentes aumentos de espacio requerido en memoria. De acuerdo con la Tabla 17 SWIFT-R introduce mayor penalización que la Replicación de procedimientos tanto en tamaño de memoria como en tiempo de ejecución. Sin embargo, el tiempo de recuperación con la Replicación de procedimientos es grande (entre 590 y 3300 ciclos de reloj para las aplicaciones analizadas), mientras que para *SWIFT-R* es despreciable. Esto se debe a que *SWIFT-R* realiza comprobaciones mucho más frecuentemente, lo que conlleva una menor latencia a cambio de una mayor penalización en las prestaciones.

A la vista de los resultados obtenidos se puede concluir que combinando la técnica *Predicción del contador de programa* con técnicas de endurecimiento software se puede alcanzar una solución con una cobertura a fallos muy elevada, disminuyendo hasta en dos órdenes de magnitud la cantidad de errores no detectados.

Software	**Tipo de error**	***Sin HM***	***SR***	***RP***	***SR+HM***	***RP+HM***	***Reducción relativa***
Mmult	***SCD(%)***	9.65%	2.92%	0.30%	2.18%	0.15%	x 113
	HANG(%)	6.94%	4.74%	4.98%	0.00%	0.00%	
	Total(%)	16.59%	7.66%	5.28%	2.18%	0.15%	
PID	***SCD(%)***	12.97%	4.73%	0.80%	2.18%	0.15%	x 15
	HANG(%)	7.77%	5.27%	5.73%	0.09%	1.25%	
	Total(%)	20.74%	10.00%	6.53%	2.28%	1.40%	
FIR	***SCD(%)***	12.39%	3.21%	0.88%	0.69%	0.15%	x 114
	HANG(%)	5.55%	5.20%	4.79%	0.01%	0.01%	
	Total(%)	17.94%	8.41%	5.67%	0.69%	0.16%	

Tabla 18: Campaña de inyección de fallos *SEU* en emulación para el sistema basado en el microprocesador *PicoBlaze* y *HM-Predicción del PC* con endurecimiento software.

Software	Tipo de error	*Sin HM*	*SR*	*RP*	*SR+HM*	*RP+HM*	*Reducción relativa*
Mmult	***SCD(%)***	0.83%	0.30%	0.05%	0.16%	0.02%	x 78
	HANG(%)	0.90%	0.61%	0.76%	0.00%	0.00%	
	Total(%)	1.73%	0.91%	0.80%	0.16%	0.02%	
PID	***SCD(%)***	1.30%	0.64%	0.05%	0.16%	0.01%	x 22
	HANG(%)	1.18%	0.93%	0.95%	0.01%	0.10%	
	Total(%)	2.48%	1.56%	1.00%	0.17%	0.11%	
FIR	***SCD(%)***	1.20%	0.56%	0.12%	0.07%	0.02%	x 114
	HANG(%)	1.04%	0.92%	0.92%	0.00%	0.00%	
	Total(%)	2.24%	1.48%	1.04%	0.07%	0.02%	

Tabla 19: Campaña de inyección de fallos *SET* en emulación para el sistema basado en el microprocesador *PicoBlaze* y *HM-Predicción del PC* con endurecimiento software.

5.4. Análisis con el microprocesador LEON3 como caso de estudio

Una vez comprobada la capacidad de la técnica *Predicción del contador de programa* para detectar errores trabajando conjuntamente con técnicas de endurecimiento software en un microprocesador de arquitectura sencilla como *PicoBlaze* se ha procedido a realizar una serie de campañas de inyección utilizando el microprocesador *LEON3* como caso de estudio y combinando técnicas de endurecimiento software con las técnicas de *Control flow checking* presentadas en este trabajo de tesis. En primer lugar se ha llevado a cabo un estudio comparativo de tres de las principales técnicas de endurecimiento software actuales utilizando el *LEON3* como plataforma común para la realización de los experimentos. Hasta ahora se tenían datos de la capacidad de detección de errores de cada una de estas técnicas utilizando sistemas basados en microprocesadores diferentes para la realización de los experimentos. Por tanto, los resultados que se aportaban eran difícilmente comparables. Con el estudio comparativo realizado en este trabajo de tesis se puede determinar la eficiencia a la hora de detectar errores de diferentes técnicas

de endurecimiento software bajo una misma plataforma y como consecuencia se puede realizar una comparación de las mismas determinando cuál de ellas presenta mejores resultados. Una vez realizado el estudio comparativo se ha procedido a realizar una campaña de inyección de fallos en un sistema basado en el microprocesador *LEON3* al que se ha añadido el *Monitor Hardware* y una combinación de las técnicas de endurecimiento software que presentan mejores resultados. El *Monitor Hardware* ha sido implementado con las técnicas *Predicción del contador de programa* y *Monitorización Dual*. Esta campaña de inyección de fallos es equivalente a la realizada en el apartado 4.5.

5.4.1. Análisis de la técnica de Monitorización de Firmas y comparación de técnicas de endurecimiento software

En este apartado se evalúan diferentes técnicas de endurecimiento software, utilizando el microprocesador *LEON3* como plataforma común, con la finalidad de comprobar su eficiencia a la hora de detectar errores en los datos.

A pesar que hay una amplia variedad de técnicas de endurecimiento software en la literatura, es difícil comparar las ventajas y desventajas de cada una de ellas basándonos en los resultados publicados. Los experimentos han sido realizados en diferentes plataformas, con diferentes microprocesadores y aplicaciones software. Además las campañas de inyección no son comparables ya que generalmente los fallos no son inyectados en la totalidad del circuito y los autores usan una amplia variedad de modelos de fallos. Muchas de las campañas de inyección únicamente consideran la inyección de *SEUs* en zonas muy concretas del circuito, como el banco de registros, contador de programa o subconjuntos de registros accesibles desde el software. En estos casos las técnicas no están completamente evaluadas ya que un fallo puede ocurrir en cualquier momento y en cualquier parte del circuito. Además, la precisión de los experimentos es cuestionable puesto que generalmente se utilizan modelos funcionales, antes de la síntesis, y por tanto no tienen en cuenta la estructura real del circuito una vez implementado. Otras técnicas están centradas en

detectar determinados tipos de errores que podrían no ser representativos si se considera una campaña de inyección más amplia. Por último, un aspecto muy importante es que cómo la mayoría de los trabajos utilizan técnicas de inyección de fallos mediante simulación, la amplitud de las campañas se limita generalmente a un conjunto de fallos pequeño, del orden de unos miles de fallos.

Para obtener resultados comparables, se han realizado diversos experimentos con las técnicas de endurecimiento software seleccionadas para el mismo procesador y la misma aplicación software. Además, gracias a la herramienta *AMUSE,* se han inyectado millones de fallos de tipo *SEU* y *SET,* cubriendo todos los posibles puntos de fallo y todos los ciclos de reloj en una implementación sintetizada del microprocesador LEON3 Los resultados obtenidos en este estudio comparativo han sido publicados en [Parr14c].

5.4.1.1. Características de los experimentos

Para llevar a cabo el estudio comparativo, se han seleccionado tres de las técnicas descritas en el aparatado 5.2: *Data duplication, Final variables* e *Inverted branches.* Estas técnicas han sido seleccionadas teniendo en cuenta la eficiencia a la hora de detectar errores y la simplicidad de las transformaciones que hay que realizar al código para poder implementarlas. Sólo se han considerado técnicas de endurecimiento software para detectar errores de datos, ya que los errores de control de flujo se detectan mediante un módulo hardware externo.

Por simplicidad, la comparación se ha realizado para una sola aplicación software, el algoritmo de ordenación Bubble sort. La versión original en C del algoritmo *Bubble sort* ha sido modificada para obtener los tres códigos endurecidos correspondientes a las distintas técnicas a estudiar: *DD* (*Data duplication*), *FV* (*Final variables*) e *IV* (*Inverted branches + Variables*). Todas las versiones fueron compiladas con GCC usando la opción de optimización –O2.

La Tabla 20 muestra el impacto sobre el rendimiento del sistema que supone implementar cada una de las técnicas de endurecimiento software.

La peor de las técnicas en cuanto al incremento que supone en el tiempo de ejecución y en el área es *Data duplication*. Esto es debido a que es la técnica que realiza más duplicaciones. Las técnicas *Final variables* e *Inverted branches + Variables* reducen el impacto sobre el rendimiento del sistema disminuyendo el número de duplicaciones y comprobaciones de consistencia.

Versión	**Ciclos de reloj**	**Incremento tiempo de ejecución**	**Incremento de área**
Original	3.404	-	-
DD	11.599	x3,4	x2,1
FV	4.792	x1,4	x1,9
IV	5.252	x1,5	x1,9

Tabla 20: Incrementos de área y tiempo de ejecución para las técnicas *Data duplication, Final variables* y *Inverted+Variables.*

La campaña de inyección de fallos ha sido realizada utilizando la herramienta *AMUSE*. Se han realizado experimentos inyectando *SEUs* y *SETs* en el microprocesador *LEON3* para evaluar la eficacia de cada una de las técnicas. La campaña de inyección de *SEUs* ha sido realizada inyectando un fallo en cada uno de los registros del microprocesador para cada uno de los ciclos de reloj de la aplicación software a evaluar. Los fallos no han sido inyectados en el banco de registros ni en las memorias, ya que ambos pueden ser fácilmente protegibles usando códigos *EDAC*. En la campaña de inyección de *SETs* se ha inyectado un fallo en cada puerta del microprocesador y en cada ciclo de reloj, con un ancho de pulso igual al ciclo de reloj. Esto podría considerarse un peor caso en cuanto al ancho de pulso. El número de fallos inyectados varía con el número de ciclos necesarios para ejecutar la aplicación software para cada una de las técnicas de endurecimiento software, llegando a inyectarse hasta 83

millones de fallos *SETs* y 21 millones de fallos *SEUs* en la técnica con mayor número de ciclos de reloj.

En un segundo conjunto de experimentos se ha considerado la utilización conjunta de las técnicas de endurecimiento software con un Monitor Hardware para la detección de errores de control de flujo. En este caso el Monitor Hardware implementa las técnicas de Monitorización de firmas y Predicción del contador, en una versión preliminar descrita en [Boya13]. El *Monitor Hardware* obtiene el código de instrucción proporcionado por la *Interfaz de traza y* calcula el valor del siguiente contador de programa así la firma correspondiente al bloque básico que se esté ejecutando. Un error se detecta si el valor calculado del contador de programa calculado es distinto al proporcionado por la *Interfaz de traza* o la firma calculada en tiempo de ejecución difiere de la firma de referencia, calculada en tiempo de compilación.

Los fallos han sido clasificados siguiendo las siguientes categorías habituales de silenciosos, *SDC*, *HANG* y detectados. En esta última categoría se ha distinguido entre fallos detectados por el Monitor Hardware, que corresponden a fallos en el flujo de control (*Det.CFC*) y fallos detectados por el software endurecido (*Det.SW*).

5.4.1.2. Resultados experimentales

Los resultados experimentales obtenidos en las campañas de inyección de fallos de tipo *SEU* y de tipo *SET* se muestran en la Tabla 21 y la Tabla 22 respectivamente. Las cuatro primeras filas muestran los resultados obtenidos en la campaña de inyección de fallos sin incluir el *Monitor Hardware* en el sistema, siendo *NH* la versión original del software sin endurecer, DD la versión endurecida mediante *Data Duplication*, *FV* la versión endurecida mediante *Final Variables* e *IV* la versión endurecida mediante *Inverted Branches + Variales*. Las últimas cuatro filas muestran los resultados obtenidos cuando se utiliza además el *Monitor Hardware* para cada una de las versiones software endurecidas anteriores.

Versión del código	Fallos inyectados	*SDC*	*HANG*	No detectados	*Det.CFC*	*Det.SW*	*Det.total*
NH	$6{,}3 \cdot 10^6$	2,92%	2,57%	5,49%	-	-	-
DD	$21{,}5 \cdot 10^6$	0,99%	3,50%	4,48%	-	2,69%	2,69%
FV	$11{,}9 \cdot 10^6$	0,59%	3,70%	4,30%	-	3,05%	3,05%
IV	$16{,}0 \cdot 10^6$	0,34%	3,32%	3,66%	-	2,75%	2,75%
NH+HM	$6{,}3 \cdot 10^6$	2,14%	0,33%	2,48%	6,36%	-	6,36%
DD+HM	$21{,}5 \cdot 10^6$	0,75%	0,54%	1,29%	8,08%	2,02%	10,10%
FV+HM	$11{,}9 \cdot 10^6$	0,41%	0,55%	0,96%	7,14%	2,22%	9,36%
IV+HM	$16{,}0 \cdot 10^6$	0,22%	0,51%	0,73%	6,37%	1,99%	8,26%

Tabla 21: Estudio comparativo de diferentes técnicas de endurecimiento software. Campaña de inyección de fallos *SEU* en emulación.

La Tabla 16 confirma que la técnica *Inverted branches + Variables* es la que alcanza mejores resultados. Considerando el caso en el que se implementan el *Monitor Hardware* y la técnica *Inverted branches+ Variables (IV+HM)* se consigue reducir el número total de errores no detectados en 3.4 veces respecto al caso en el que se implementa únicamente el *Monitor Hardware*, sin endurecimiento software (*NH+HM)* y 7.5 veces respecto a los experimentos sin endurecimiento de ningún tipo (*NH*). La mejora en los experimentos utilizando la técnica *Inverted branches+Variables* respecto los experimentos que implementan la técnica *Final Variables* se debe principalmente a la detección de errores en las condiciones de salto de las instrucciones de salto condicionales. Si comparamos las técnicas *Data duplication y Final* variables se llega a la conclusión de que la técnica *Final variables* alcanza mejores resultados. Sin embargo, como la técnica *Data duplication* realiza más comprobaciones de variables, se reduce la posibilidad de tener errores latentes. Los errores latentes son errores que no afectan a la ejecución actual pero que pueden manifestarse en el futuro.

Teniendo en cuenta los errores latentes ambas técnicas alcanzan resultados muy similares.

Como regla general las técnicas de endurecimiento software detectan la mayoría de los errores *SDC*, mientras que las técnicas de *Control flow checking* detectan la mayoría de los errores *HANG*. La cantidad de errores *HANG* se ve incrementada en las versiones con endurecimiento software respecto a las que no están endurecidas. Esto se debe principalmente a que se produce un incremento de la longitud del código y un incremento del número de cambios en el flujo de programa al implementar las comprobaciones software. Otro efecto que puede observarse en la Tabla 16 es que el porcentaje de errores detectados mediante las técnicas software se ve reducido al introducir el *Monitor Hardware* en los experimentos. Esta reducción se debe a que hay errores que pueden ser detectados por ambas técnicas, puesto que los errores no pueden dividirse fácilmente en errores de datos y errores en el flujo de programa. Por ejemplo, en los saltos condicionales se evalúan condiciones relativas a los datos que afectan al flujo de programa. Generalmente el *Monitor Hardware* reacciona antes y clasifica el error como *Det.CFC*, aunque eventualmente pudiera ser detectado también por alguna de las comprobaciones software.

Por otro lado, también se puede observar cómo el porcentaje de errores detectados por el *Monitor Hardware* es mayor en los casos en los que se ha combinado con técnicas de endurecimiento software. Esto se debe a varias razones. Algunos errores que son detectados por el *Monitor Hardware* no afectan realmente a la ejecución actual, pero podrían quedar almacenados en el sistema como errores latentes. La capacidad de detectar este tipo de errores es muy importante para alcanzar un sistema robusto. En otros casos, hay errores que pueden producir un falso positivo, es decir, que son detectados pero no tienen ningún efecto en el resultado final de la ejecución. La consecuencia de un falso positivo es una acción de recuperación innecesaria, con la consiguiente pérdida de prestaciones. Sin embargo, dado que en la mayoría de las aplicaciones los fallos ocurren con una frecuencia muy baja, esta pérdida de prestaciones es despreciable.

En la Tabla 22 se muestran los resultados obtenidos en la campaña de inyección de fallos de tipo *SET*. En esta campaña de inyección se llegan a inyectar hasta 83 millones de fallos para la aplicación software de mayor duración. Se ha seleccionado un ancho de pulso igual a la duración del ciclo de reloj con la intención de obtener unos resultados que puedan ser fácilmente comparables con los resultados obtenidos en la campaña de inyección de fallos *SEU*. De hecho, se obtienen resultados similares en todas las categorías.

Los resultados para en la inyección de *SETs* siguen el mismo patrón que los resultados obtenidos en la inyección de *SEUs*. La técnica *Inverted branches+Variables* es también la que obtiene los mejores porcentajes de errores detectados. Considerando el caso en el que se implementan el *Monitor Hardware* y la técnica *Inverted branches+ Variables* (*IV+HM*) se consigue reducir el número total de errores no detectados en 6.4 veces respecto al caso en el que se implementa únicamente el *Monitor Hardware,* sin endurecimiento software (*NH+HM*) y 12.8 veces respecto a los experimentos sin endurecimiento de ningún tipo (*NH)*.

Versión del código	**Fallos inyectados**	***SDC***	***HANG***	**No detectados**	***Det.CFC***	***Det.SW***	***Det.total***
NH	$24{,}4 \cdot 10^6$	3,06%	2,34%	5,39%	-	-	-
DD	$83{,}3 \cdot 10^6$	0,87%	2,69%	3,56%	-	2,50%	2,50%
FV	$46{,}0 \cdot 10^6$	0,66%	3,48%	4,14%	-	3,50%	3,50%
IV	$62{,}2 \cdot 10^6$	0,31%	3,11%	3,42%	-	3,29%	3,29%
NH+HM	$24{,}4 \cdot 10^6$	2,59%	0,12%	2,71%	4,95%	-	4,95%
DD+HM	$83{,}3 \cdot 10^6$	0,71%	0,22%	0,93%	4,81%	2,19%	7,00%
FV+HM	$46{,}0 \cdot 10^6$	0,51%	0,25%	0,76%	5,80%	3,08%	8,88%
IV+HM	$62{,}2 \cdot 10^6$	0,21%	0,21%	0,42%	5,42%	2,87%	8,29%

Tabla 22: Estudio comparativo de diferentes técnicas de endurecimiento software. Campaña de inyección de fallos *SET* en emulación.

Ninguna de las técnicas de endurecimiento software analizadas en este trabajo alcanza una detección de errores total. Aparte de las posibles ineficiencias de las técnicas empleadas, esto es debido principalmente a la complejidad interna del microprocesador *LEON3,* así como a algunas deficiencias observadas en el proceso de compilación de las versiones software endurecidas. Respecto al primer punto, cabe destacar que es prácticamente imposible separar el núcleo del procesador *LEON3* de algunos componentes que no se pueden proteger mediante las técnicas empleadas, como los controladores de bus o los puertos. La protección de estos elementos debe hacerse por otros medios que están fuera del objeto de esta tesis. En cuanto al proceso de compilación, hay que tener en cuenta que algunas duplicaciones pueden ser optimizadas por el compilador *GCC*. Aunque se ha comprobado que la compilación realizada preserva las duplicaciones de variables en la mayoría de los casos, no se ha podido verificar este requisito completamente. La solución a este problema es implementar las técnicas de endurecimiento software a nivel de código ensamblador, pero esto hubiera requerido un esfuerzo de desarrollo muy grande que está también fuera del propósito de esta tesis.

5.4.2. Análisis de la Monitorización Dual con software endurecido

El tercer conjunto de experimentos presentados en este capítulo tienen por objeto evaluar la técnica de *Monitorización Dual* en combinación con las técnicas de endurecimiento software.

Mediante la realización de estos experimentos se pretende completar el análisis descrito en el apartado 4.5. En dicho análisis se llegaba a la conclusión de que las técnicas de *Control flow checking* deben trabajar en combinación con técnicas que tengan la capacidad de detectar errores en los datos para poder alcanzar una cobertura a fallos importante.

Los resultados experimentales alcanzados en esta campaña de inyección han sido publicados en [Parr14b].

5.4.2.1. Características de los experimentos

Las campañas de inyección de fallos son similares a las que se describen en el apartado 4.5 para el mismo caso pero con versiones de software no endurecidas. Se han utilizado las mismas aplicaciones software (*BBS, Mmult y AES*) y el mismo *Monitor Hardware*.

Para la realización de los experimentos se ha utilizado la herramienta *AMUSE,* inyectando fallos de tipo *SEU* y de tipo *SET* con el mismo grado de completitud que se utilizó para el software no endurecido. Debido a que el software endurecido tiene un mayor tiempo de ejecución, el número de fallos inyectados aumenta significativamente, llegando hasta 27 millones de fallos de tipo *SEU* y 350 millones de fallos de tipo *SET* en el caso de la aplicación Mmult.

Para llevar a cabo el endurecimiento software se ha utilizado la técnica *Inverted Branches + Variables*. No obstante, a partir de la experiencia obtenida en los experimentos anteriores, se han simplificado los procesos de comprobación. De forma más concreta, en las aplicaciones software se ha realizado la duplicación de todas las variables, pero únicamente se han realizado comprobaciones de consistencia sobre las variables más relevantes, incluyendo las variables de salida.

Adicionalmente se ha realizado una tercera tanda de experimentos inyectando fallos en el banco de registros. El objetivo de esta campaña de inyección es evaluar la capacidad del *Monitor Hardware* para detectar también errores en el banco de registros, aunque los errores en el banco de registros son errores de datos y cabe esperar que el porcentaje detectado sea bajo. El banco de registros del *LEON3* consiste en dos módulos de *RAM* que implementan las ocho ventanas de registros y los ocho registros globales. Siguiendo la metodología habitualmente utilizada en esta tesis, se han inyectado fallos de tipo *SEU* de manera exhaustiva en todos los bits de dichos módulos *RAM* y en todos los ciclos de reloj de cada ejecución.

5.4.2.2. Resultados experimentales

Los resultados obtenidos en la campaña de inyección de fallos se muestran en la Tabla 23 , la Tabla 24 y en la Tabla 25 para las aplicaciones software *Bubble Sort, Multiplicación de matrices* y *AES* respectivamente. Finalmente en la Tabla 26 se muestran los resultados obtenidos en la campaña de inyección realizada sobre el banco de registros.

Al igual que en los experimentos reportados en el apartado 4.5, los registros del microprocesador se han dividido en dos grupos. El grupo de registros I está formado por el contador de programa y el registro de instrucción de todas las etapas del pipeline, con 346 registros en total. El grupo de registros II incluye el resto de registros del microprocesador, y está formado por 1505 registros. Las tres primeras filas de la Tabla 23 , Tabla 24 y Tabla 25 muestran los resultados de los experimentos de inyección de fallos *SEUs* en el grupo de registros I, grupo de registros II y en la totalidad de registros respectivamente. La última fila muestra los resultados obtenidos en la inyección de fallos *SETs*. De izquierda a derecha se muestran los fallos inyectados, los errores observados, el número de errores de tipo *SDC*, el número de errores de tipo *HANG* y los errores detectados por el *Monitor Hardware* y el *software* endurecido conjuntamente.

La Tabla 23 muestra los resultados obtenidos en la campaña de inyección en la que se utiliza el algoritmo de ordenación *Bubble sort* como caso de estudio. En este caso, la aplicación necesita un total de 8663 ciclos de reloj para ser ejecutada y el total de fallos inyectados asciende a 16 millones de *SEUs* y 205 millones de *SETs*. Al igual que sucedía en los experimentos realizados sin endurecimiento software (apartado 4.5) se detecta el 100% de los errores en el grupo de registros I y muchos de los errores del grupo de registros II. Aunque el grupo de registros I es mucho menor que el grupo de registros II, contiene aproximadamente la mitad de los errores observados durante la ejecución de la aplicación software. Esto es debido a que el contador de programa y el código de instrucción son registros muy críticos. De hecho, los errores que se producen en el grupo de registros I son en su totalidad errores que afectan al flujo de programa

[Vemu07]. Combinando el *Monitor Hardware* con la técnica de endurecimiento software se consigue alcanzar una detección del 92.0% en los experimentos de inyección de fallos de tipo *SEU y* un 95.2% en los experimentos de inyección de fallos de tipo *SET*.

Elementos	Fallos inyectados	Errores observados	SDC	Hang	Errores detectados
PC & IR (I)	2,997 M	767.712	0	0	767.712 (100%)
Otros Regs. (II)	13,038 M	813.107	81.996 (10,1%)	45.195 (5,6%)	685.916 (84,4%)
Regs. (todos)	16,035 M	1.580.819	81.996 (5,2%)	45.195 (2,9%)	1.453.628 (92,0%)
Lógica comb. (SETs)	205,233 M	1.881.373	78.992 (4,2%)	10.448 (0,6%)	1.791.933 (95,2%)

Tabla 23: Resultados de inyección de fallos con Monitorización *Dual* y endurecimiento software (*Bubble Sort*)

Los resultados de la inyección de fallos utilizando la aplicación software *Multiplicación de matrices* como caso de estudio se muestran en la Tabla 24. La *Multiplicación de matrices* es una aplicación más compleja que el algoritmo de ordenación *Bubble sort* y requiere 14788 ciclos de reloj para completar el programa. Los resultados que se han obtenidos son muy similares a los obtenidos con el algoritmo de ordenación *Bubble sort*. De nuevo, se detectan todos los errores que se producen en el grupo de registros I y un conjunto importante de errores en el grupo de registros II. En este caso, al llevar a cabo el endurecimiento software sobre la aplicación caso de estudio e implementando el *Monitor Hardware* en el sistema se consigue alcanzar una detección del 93.4% en los experimentos de inyección de fallos de tipo SEU y un 96.9% en los experimentos de inyección de fallos de tipo SET.

Elementos	Fallos inyectados	Errores observados	SDC	Hang	Errores detectados
PC & IR (I)	5,11 M	1.173.644	0	0	1.173.328 (100%)
Otros Regs. (II)	22,3 M	1.586.301	65.174 (4,1%)	117.046 (7,4%)	1.404.397 (88,5%)
Regs. (todos)	27,4 M	2.759.945	65.174 (2,4%)	117.046 (4,2%)	2.577.725 (93,4%)
Lógica comb. (SETs)	350,3 M	3.921.304	69.807 (1,8%)	50.633 (1,3%)	3.800.864 (96,9%)

Tabla 24: Resultados de inyección de fallos con Monitorización Dual y endurecimiento software (*Multiplicación de matrices*)

Los resultados obtenidos en la tercera aplicación, *AES,* se muestran en la Tabla 25. El algoritmo de ordenación *AES* es la aplicación que presenta un tamaño de código mayor. Sin embargo, la duración de la ejecución no supera los 6564 ciclos de reloj para la versión endurecida. Una vez más se obtienen resultados muy similares, alcanzando una detección del 92.8% de los errores en los experimentos de inyección de fallos de tipo *SEU* y del 93.0% en los experimentos de inyección de fallos de tipo *SET*.

En los tres casos de aplicación hay un pequeño porcentaje de errores que no se consigue detectar por ninguna de las técnicas implementadas. Estos errores no detectados se corresponden en su mayoría con fallos inyectados fuera del núcleo del procesador. Por ejemplo, un error en el controlador de memoria o en el controlador de bus puede afectar de manera común a las variables duplicadas. Estos errores escapan de las capacidades de detección de las técnicas descritas en este trabajo y deberían ser tratados por otro tipo de técnicas específicas, que están fuera del ámbito del trabajo desarrollado en esta tesis doctoral.

Elementos	Fallos inyectados	Errores observados	SDC	Hang	Errores detectados
PC & IR (I)	2,271M	1.150.973	0	0	1.150.973 (100%)
Otros Regs. (II)	9,879M	762.482	107.390 (14,1%)	30.487 (4,0%)	624.605 (81,9%)
Regs. (todos)	12,150M	1.913.455	107.390 (5,6%)	30.487 (1,6%)	1.775.578 (92,8%)
Lógica comb. (SETs)	155,511M	1.336.025	78.167 (5,9%)	2.999 (0,2%)	1.254.859 (93,9%)

Tabla 25: Resultados de inyección de fallos con Monitorización Dual y endurecimiento software (*AES*)

Por último, se ha llevado a cabo una campaña de inyección de fallos en el banco de registros del microprocesador *LEON3*. Los bloques de *RAM* que implementan el banco de registros únicamente están protegidos por el *Monitor Hardware* y la técnica de endurecimiento software. Los resultados de la campaña de inyección en el banco de registros para las diferentes aplicaciones software se muestran en la Tabla 26. Al igual que en los anteriores experimentos se ha hecho una inyección exhaustiva de fallos de tipo SEU para todos los bits de los bloques de RAM.

Las tres primeras filas de la Tabla 26 muestran los resultados para las diferentes aplicaciones software originales, es decir, añadiendo al sistema únicamente el *Monitor Hardware*. Aunque los fallos inyectados únicamente producen errores en los datos, el *Monitor Hardware* es capaz de detectar el 30.8% de los errores en el algoritmo de ordenación *Bubble sort,* el 42.8% en la *Multiplicación de matrices* y el 38.0% en el algoritmo de encriptación *AES*. Estos errores son detectados en su mayoría por el tratamiento de excepciones y la condición del *timeout* que implementa el *Monitor Hardware*. Las últimas tres filas de la Tabla 26 muestran los resultados con las versiones endurecidas de cada una de las aplicaciones software. En este caso el porcentaje de detección crece hasta alcanzar el 93.3% en el

algoritmo de ordenación *Bubble sort,* el 92.4% en la *Multiplicación de matrices* y el 97.3% en el algoritmo de encriptación *AES*.

La razón por la que no se detectan todos los errores es porque el compilador optimiza el código eliminando parte de las duplicaciones realizadas durante el endurecimiento software. Se puede mejorar la capacidad de detección de errores disminuyendo el nivel de optimización del compilador a expensas de incrementar el tiempo de ejecución. Por ejemplo, se consigue alcanzar una detección de errores del 99.3% en el caso de la aplicación *Multiplicación de matrices* reduciendo el nivel de optimización del compilador a –O0, tal y como se muestra en la última fila de la Tabla 26.

Caso	Fallos inyectados	Errores observados	*SDC*	*HANG*	Errores detectados
BBS HM	29,628M	509.955	310.197 (60,8%)	42.467 (8,3%)	157.291 (30,8%)
Mmult HM	53,469M	1.413.124	580.597 (41,1%)	227.806 (16,1%)	604.721 (42,8%)
AES HM	71,708M	1.414.901	844.203 (59,7%)	32.470 (2,3%)	538.228 (38,0%)
BBS HM+SW	75,403M	2.626.626	175.623 (6,7%)	0 (0%)	2.451.003 (93,3%)
Mmult HM+SW	128,715M	1.510.704	49.016 (3,2%)	65.885 (4,4%)	1.395.803 (92,4%)
AES HM+SW	107,538M	3.479.705	22.668 (0,7%)	72.626 (2,1%)	3.384.411 (97,3%)
Mmult HM+SW –O0	284,621M	1.976.912	7.864 (0,4%)	6.040 (0,3%)	1.963.008 (99,3%)

Tabla 26: Resultados de inyección de fallos en el banco de registros para las diferentes aplicaciones software utilizadas durante los experimentos.

A la vista de los resultados obtenidos se puede concluir que la *Monitorización Dual* es capaz de detectar la totalidad de los errores que afectan al flujo de control, independientemente de si el software se ha endurecido o no. Además, complementando dichas técnicas con técnicas de endurecimiento software se consigue alcanzar una solución eficiente contra todos los fallos que puedan ocurrir en el microprocesador, con un impacto sobre el área y el rendimiento del sistema reducido.

5.5. Comparación con otras técnicas híbridas

Para terminar este capítulo, en este apartado se realiza un análisis comparativo de los métodos utilizados y los resultados obtenidos respecto a otras técnicas híbridas existentes en la literatura.

En primer lugar, es importante destacar que las técnicas propuestas en esta tesis no requieren realizar modificaciones en el software para llevar a cabo el control del flujo de programa. Por tanto, el incremento del tiempo de ejecución únicamente se debe al endurecimiento software. Esta es una ventaja sustancial, ya que la gran mayoría de las técnicas de detección de errores de control de flujo introducen una penalización significativa en el tiempo de ejecución. Por ejemplo, [Azam11a] introduce hasta un 61% de penalización utilizando técnicas basadas en firmas y [Azam13] hasta un 34% usando aserciones.

La penalización en el tiempo de ejecución debida al endurecimiento software depende en gran medida de la técnica utilizada. En las implementaciones realizadas es similar a otras técnicas híbridas existentes como las descritas en [Bern06], [Azam11a] o Azam13]. Esta penalización varía también con la frecuencia con la que se realizan comprobaciones. Así, la técnica *SWIFT-R* tiene una mayor penalización que la Replicación de procedimientos, pero a cambio reduce la latencia de la detección. El incremento en el tamaño del código que supone la implementación de las técnicas propuestas es también menor o similar a otras técnicas existentes.

Un ejemplo extremo del compromiso entre tiempo de ejecución y latencia de detección es el que se muestra en [Port12]. En este caso, se utiliza una duplicación completa del software, repitiendo completamente la ejecución, o bien una duplicación completa del hardware, utilizando dos procesadores que ejecutan la aplicación de manera redundante. Esto proporciona una alta tasa de detección, pero a cambio de una latencia muy alta, ya que hay que esperar a la finalización de la ejecución para poder determinar si ha habido algún error. En contraposición, las técnicas de detección de errores propuestas en esta tesis permiten detectar todos los errores de control de flujo sin ninguna penalización y con una latencia mínima. Como se ha demostrado en este capítulo, estas técnicas son compatibles con cualquier técnica de detección de errores de datos implementada en software.

Otro de los puntos que es importante destacar de este trabajo es el método de inyección que se ha utilizado. Mientras que en el trabajo propuesto en [Bern06] se realiza una inyección de fallos vía software, y en los trabajos propuestos en [Azam11a], [Azam13] y [Port12] se inyecta directamente en las señales del microprocesador descrito en VHDL, en este trabajo los fallos son inyectados en cada uno de los nodos de la *netlist* sintetizada y considerando los retrasos reales estimados por la herramienta de síntesis, lo que aporta una mayor precisión. Este análisis tan completo dota a los resultados de una precisión muy superior.

La inyección de fallos vía *software* está inherentemente limitada a los registros accesibles por el usuario. Por este motivo, sólo se inyectan fallos en zonas muy concretas del microprocesador, y por tanto, únicamente se evalúa la capacidad de detección de errores en estas zonas [Nico04], [Bern06] y [Chey00]. Típicamente, la inyección de fallos se limita al contador de programa y el registro de instrucción, el banco de registros y las memorias de datos y de programa. Sin embargo, en los procesadores modernos, con varias etapas de *pipeline*, existen numerosos registros que no son accesibles vía software, por lo que la inyección de fallos con esta técnica no es suficientemente representativa. Cuando se utiliza un modelo funcional en

VHDL es posible acceder a todos los registros e incluso utilizar modelos de fallos elaborados. Sin embargo, dado que se abstrae la implementación real del microprocesador, la precisión es escasa. En particular, los fallos de tipo *SET* no se pueden modelar con precisión, ya que no se tienen en cuenta los retardos.

Por último, gracias al sistema de emulación AMUSE, el número de fallos inyectados en los experimentos es muchísimo mayor que en los trabajos previos. Por ejemplo, los resultados de [Port12] se han obtenido inyectando 10.000 fallos. Esta cifra se incrementa en otros casos hasta 50.000 fallos [Azam11a] o 100.000 fallos [Bern06],[Azam13]. En contraposición, en este trabajo de tesis las campañas de inyección son exhaustivas en la inyección de SEUs, ya que se ha inyectado en todos los biestables y en todos los ciclos de reloj, alcanzando las decenas de millones de fallos inyectados. En cuanto a la inyección de *SETs*, se han inyectado varios fallos en cada ciclo de reloj para todas las puertas lógicas, alcanzando los centenares de millones de fallos en cada campaña, además de tener en cuenta los retardos.

Si tenemos en cuenta la capacidad de detección de errores, al igual que en los trabajos propuestos en [Bern06], [Azam11a], [Azam13] y [Port12], se alcanza una detección cercana al 100%. Sin embargo las campañas de inyección de fallos llevadas a cabo en este trabajo son mucho más completas. Cabe destacar que cuando se afirma que la técnica de *Monitorización Dual* detecta el 100% de los fallos de control de flujo, esta afirmación no es de carácter estadístico, sino que se ha probado de manera exhaustiva para las aplicaciones analizadas.

5.6. Resumen y conclusiones

En este capítulo se ha analizado la capacidad de detección de errores de las técnicas propuestas en esta tesis cuando se combinan con técnicas de endurecimiento software para errores de datos. Mediante esta combinación de técnicas se ha podido alcanzar una solución de compromiso entre la tolerancia a fallos y el impacto sobre el rendimiento del sistema, tal y como se proponía en los objetivos de esta tesis doctoral.

Al igual que los experimentos realizados en el capítulo 4, en los que no se llevaba a cabo un endurecimiento software, se han realizado experimentos utilizando los microprocesadores *LEON3* y *Picoblaze,* pero en este caso únicamente se han realizado experimentos de emulación en FPGA, que son mucho más precisos, dada la amplitud de las campañas de inyección de fallos que se pueden realizar.

En primer lugar se han llevado a cabo los experimentos en el microprocesador *Picoblaze,* en los que se ha implementado la técnica de *Predicción del contador de programa* combinada con dos técnicas de endurecimiento software diferentes. Al tratarse de un microprocesador sencillo, la técnica de *Predicción del contador de programa* resulta suficientemente efectiva para la detección de errores de control de flujo. Es importante destacar que esta técnica puede ser implementada sin necesidad de almacenar información calculada en tiempo de compilación, y que por tanto, los recursos necesarios para su implementación son reducidos. Los resultados obtenidos demuestran que utilizando las técnicas hardware propuestas en esta tesis doctoral junto con técnicas software para errores de datos se puede reducir la tasa de errores total hasta en dos órdenes de magnitud.

En segundo lugar se han analizado varias técnicas de endurecimiento software, así como su combinación con la técnica de Monitorización de Firmas para la detección de errores de control de flujo. A pesar de que existe una amplia variedad de técnicas de endurecimiento software, es muy difícil compararlas entre sí, ya que los experimentos han sido realizados utilizado diferentes microprocesadores, distintos modelos de fallos y distintos métodos de inyección de fallos. Se han seleccionado tres de las técnicas de endurecimiento software más recientes entre las existentes en la literatura, las cuales han sido evaluadas bajo la misma plataforma y utilizando el mismo método de inyección. Gracias a la utilización de la herramienta *AMUSE* se han podido llevar a cabo campañas de inyección de millones de fallos de tipo *SEU* y de tipo *SET.* Estas extensas y exhaustivas campañas de inyección de fallos han permitido un análisis mucho más completo, que no podría ser

considerado mediante otros métodos de inyección más sencillos, y realizar una comparativa de las diferentes técnicas de endurecimiento software seleccionadas.

En tercer lugar, se han analizado las técnicas de *Predicción del contador de programa* y *Monitorización Dual* en combinación con las técnicas de endurecimiento software que obtuvieron mejores resultados. Los resultados que se han obtenido han sido muy satisfactorios, detectando la totalidad de los errores de flujo y un porcentaje muy elevado de todos los errores, en torno al 95%. La mayoría de los errores que no son detectados son errores que han sido inyectados fuera del núcleo del microprocesador y que están fuera del ámbito de las técnicas propuestas. Estos errores deben de ser detectados utilizando otro tipo de técnicas. Además hay que destacar que para realizar el control del flujo de programa no ha sido necesario almacenar información de la ejecución calculada en tiempo de compilación, lo que supone una reducción del área necesaria para implementar las técnicas propuestas. Por otra parte, estas técnicas no introducen tampoco ninguna penalización en las prestaciones. Se ha conseguido, por tanto, una alta tasa de detección de errores y minimizar el impacto en el rendimiento del sistema, tal y como se pretendía al iniciar la tesis doctoral.

En comparación con otras técnicas híbridas existentes, cabe destacar que las técnicas propuestas proporcionan una alta tasa de detección de errores, con un impacto reducido en el incremento de hardware, mientras que el impacto sobre las prestaciones se ha reducido puesto que la detección de fallos de control de flujo se realiza sin ninguna penalización.

Teniendo en cuenta los resultados obtenidos y presentados en este capítulo se puede concluir que al combinar las técnicas de *Control flow checking*, desarrolladas en este trabajo de tesis, con técnicas de endurecimiento software para detectar errores en los datos, se consigue alcanzar una solución completa frente a los posibles fallos que se pueden dar en un microprocesador. Además el impacto sobre el rendimiento y el área del sistema que supone implementar las diferentes técnicas es reducido, lo que supone un valor añadido al trabajo realizado.

CAPÍTULO 6

CONCLUSIONES

6. Conclusiones

Los microprocesadores constituyen el núcleo de los sistemas digitales actuales y están presentes en una infinidad de aplicaciones esenciales para nuestra sociedad, incluyendo aplicaciones críticas en las que un fallo puede provocar un error de cómputo o la pérdida de control de un sistema con consecuencias catastróficas. La evolución de la tecnología ha permitido disponer de microprocesadores con elevadas prestaciones a muy bajo coste, pero a la vez ha aumentado la susceptibilidad a los errores transitorios. Por este motivo, el uso de técnicas de tolerancia a fallos que permitan detectar o corregir los errores transitorios es imprescindible actualmente en un número creciente de aplicaciones.

De entre las técnicas existentes para la detección de errores en microprocesadores, destacan actualmente las soluciones híbridas, las cuales tratan de aunar las ventajas que presentan de las aproximaciones hardware y software. En este sentido, la solución más efectiva consiste en utilizar un módulo hardware externo para detectar los errores de control de flujo junto con un software endurecido para detectar errores de datos.

En esta tesis se han propuesto tres técnicas de detección de errores de control de flujo en microprocesadores. Estas técnicas tienen en común que se pueden implementar de manera no intrusiva en un módulo hardware externo y pueden combinarse eficientemente con técnicas de software para obtener una alta capacidad de detección de errores. Un elemento novedoso adicional es la utilización de la interfaz de traza como medio de observación de la ejecución. Aunque la utilización de la interfaz de traza ya se había propuesto anteriormente, en esta tesis se profundiza en sus posibilidades, aplicándola a la detección de errores de control de flujo con mínima latencia y sin penalización de las prestaciones.

La primera técnica propuesta es la de Predicción del Contador de Programa. Esta técnica se basa en calcular el valor del contador de programa a partir del valor anterior del mismo y el código de instrucción actual obtenidos desde la interfaz de traza. En esencia, se explota una

información implícita en el hecho de que la ejecución de las instrucciones en un microprocesador sigue un orden secuencial que sólo puede ser alterado por determinadas instrucciones. Esta técnica es capaz de detectar un subconjunto de errores, concretamente los que afectan al contador de programa. Sin embargo, es muy eficiente para detectar este tipo de errores y además puede ser implementada con un conjunto de recursos muy reducido, ya que no precisa almacenar información en tiempo de compilación. Para el caso de un microprocesador sencillo como Picoblaze, esta técnica es capaz de detectar la mayoría de los errores de tipo HANG. Por otra parte, esta técnica es complementaria a las otras dos técnicas descritas en la tesis, Monitorización de firmas y Monitorización dual, y necesaria para que éstas alcancen elevadas tasas de detección de errores de control de flujo, como se puede observar en los resultados de los experimentos realizados.

La técnica de Monitorización de Firmas verifica la ejecución de un bloque de instrucciones mediante el cálculo de una firma online. Cada bloque tiene asignada una firma de referencia que es calculada en tiempo de compilación. Al final de la ejecución de cada bloque se comparan la firma de referencia con la calculada durante la ejecución para determinar si se ha producido un error en la ejecución de dicho bloque. Esta técnica presenta dos mejoras respecto a técnicas basadas en el cálculo de firmas propuestas por otros autores. Pon un lado se consigue reducir el tamaño de la memoria necesaria para almacenar las firmas de referencia utilizando la tabla CFC-ST. Se han propuesto dos métodos de almacenamiento y acceso a la tabla CFC-ST, un método estático y un método dinámico. En ambos casos se consigue reducir el impacto sobre el rendimiento del sistema al reducir el tamaño de la tabla necesaria para almacenar las firmas.

Los resultados experimentales demuestran que la técnica de Monitorización de Firmas es capaz de detectar todos aquellos errores que afectan al código de instrucción siempre que se almacenen todas las firmas de referencia calculadas en tiempo de compilación, bajando levemente la capacidad de detección si la tabla CFC-ST no puede almacenar todas las

firmas. Trabajando en conjunto con la técnica Predicción del contador de programa alcanza un 100% de detección de errores que afectan al flujo de programa. Por otra parte, el método dinámico permite trabajar con una tabla CFC-ST de tamaño limitado y optimizar la capacidad de detección de errores, lo que resulta especialmente indicado para aplicaciones de software grandes. En general, el método dinámico produce resultados próximos a la detección completa y es más efectivo que el método estático.

En tercer lugar, la técnica de Monitorización Dual realiza la monitorización de la ejecución utilizando dos puntos de observación diferentes, la interfaz de traza y el bus de memoria. A través de estos dos puntos de observación se obtienen el contador de programa y el código de instrucción en la etapa de búsqueda y justo después de la etapa de ejecución, permitiendo la comparación de estos valores antes y después de que se haya ejecutado cada instrucción. Combinando esta técnica con Predicción del Contador de Programa se alcanza un 100% de detección de errores de control de flujo y un porcentaje cercano al 70% cuando se consideran todos los posibles errores. En combinación con técnicas de endurecimiento de software para errores de datos, el porcentaje de detección alcanza valores en torno al 95%. La mayoría de los errores que no son detectados son errores que han sido inyectados fuera del núcleo del microprocesador y que están fuera del ámbito de esta tesis.

Una de las ventajas principales de las técnicas de Predicción del Contador de Programa y de Monitorización Dual es que consiguen eliminar la necesidad de almacenar información calculada en tiempo de compilación, y por tanto, reducen de una manera drástica los recursos necesarios para su implementación y el impacto sobre el rendimiento del sistema. Esta es una aportación notable que no estaba presente en las técnicas de monitorización anteriores. En el caso de la Monitorización Dual no sea factible, debido a que requiere el acceso a dos puntos diferentes de observación, la técnica de Monitorización de Firmas propuesta en esta tesis proporciona una solución optimizada que permite obtener buenos resultados con un uso limitado de recursos de almacenamiento.

Por último, es importante destacar que todas las técnicas propuestas se han probado de manera mucho más amplia que las técnicas existentes. En particular, gracias a la herramienta de inyección de fallos AMUSE, la inyección de fallos de tipo SEU se ha realizado de manera exhaustiva y se han podido realizar campañas de inyección muy grandes, del orden de cientos de millones de fallos, para fallos de tipo SET.

A la vista de los resultados se puede concluir que las técnicas presentadas en esta tesis protegen de una manera eficiente frente a errores en el flujo de programa alcanzando una buena solución de compromiso entre la cobertura a fallos y el impacto sobre el rendimiento del sistema. Además, todas las técnicas pueden implementarse de manera no intrusiva, lo que las hace susceptibles de ser utilizadas con cualquier procesador sin necesidad de modificarlo, siempre que se disponga de una interfaz de traza elemental.

6.1. Trabajos futuros

En esta tesis doctoral se han propuesto varias técnicas de detección de errores de control de flujo y se ha demostrado que son altamente efectivas, permitiendo una detección total de este tipo de errores con un coste reducido en hardware. Sin embargo, para una solución completa, es necesario utilizar técnicas específicas para la detección de errores de datos. Las soluciones existentes para este propósito se basan actualmente en técnicas de software.

Mientras que la detección de errores de control se puede realizar mediante el Monitor Hardware de manera muy eficiente, para la detección de errores de datos es necesario seguir buscando soluciones menos costosas. En principio, las soluciones hardware son mucho más difíciles, puesto que las comprobaciones que es necesario realizar para detectar errores de datos son específicas de cada programa. De nuevo, el aprovechamiento de infraestructuras ya existentes en los procesadores modernos puede ser clave para abordar este problema. En particular, una línea prometedora consiste en la utilización de extensiones SIMD (Single

Instruction Multiple Data). Estas extensiones, que están disponibles en las arquitecturas de los principales procesadores actuales como ARM (NEON) o Intel (SSE), están pensadas para acelerar cálculos idénticos que se han de realizar sobre varios datos. Por tanto, se podrían aprovechar para realizar cálculos redundantes y comparar los resultados para detectar errores de datos.

Las técnicas propuestas en esta tesis se centran exclusivamente en el microprocesador. Sin embargo, en los sistemas actuales existen elementos que son inseparables del mismo, como los controladores de memoria, los buses o los periféricos. La detección de errores en estos elementos se debe realizar por otros medios que están fuera del ámbito de esta tesis. La extensión a estos componentes de las técnicas propuestas, así como la combinación con otras técnicas existentes, es un tema de interés de cara a la consecución de sistemas completos tolerantes a fallos.

Para finalizar, es conveniente realizar ensayos bajo radiación como demostración definitiva de las técnicas propuestas.

Bibliografía

[Alkh99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, Vol.10, NO.6, June 1999.

[Ashr07] Ashraf M. El-Attar; Gamal Fahmy, "An Improved Watchdog Timer to Enhance Imaging System Reliability In The Presence Of Soft Errors", IEEE International Symposium on Signal Processing and Information Technology, pp.1100-1104, 2007.

[Azam10] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt, "The limitations of software signature and basic block sizing in soft error fault coverage," in Proc. IEEE Latin-American Test Workshop, 2010.

[Azam11a] J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt, "Detecting SEEs in microprocessors through a non-intrusive hybrid technique," IEEE Trans. Nucl. Sci., vol. 58, no. 3, pp. 993–1000, Jun. 2011.

[Azam11b] J. R. Azambuja, S. Pagliarini, L. Rosa, and F. L. Kastensmidt, "Exploring the limitations of software-only techniques in SEE detection coverage," J. Electron. Test., no. 27, pp. 541–550, 2011.

[Azam12] J.R. Azambuja, S. Pagliarini; M. Altieri, F.L. Kastensmidt; M. Hubner; J. Becker; G. Foucard; R. Velazco, "A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware" IEEE Transactions on Nuclear Science, Volume: 59 , Issue: 4 , Part: 1, 2012, pp: 1117 – 1124.

[Azam13] J. R. Azambuja, M. Altieri, J. Becker, F. L. Kastensmidt. "HETA: Hybrid Error-Detection Technique Using Assertions". IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2805-2812, Aug. 2013.

[Bens00] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications", Proceedings of the IEEE International Conference on Dependable Systems and Networks, 2000, pp. 71-78.

[Bens03] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "A Watchdog Processor to Detect Data and Control Flow Errors", 9th IEEE International On-Line Testing symposium, Kos, Greece, July 7-9, 2003, pp. 144-148.

[Berg09] S. Bergaoui, R. Leveugle, "ISDM: An Improved Control Flow Checking Approach with Disjoint Signature Monitoring", Proc. 24th Conference on Design of circuits and Integrated Systems (DCIS), 2009.

[Bern06] P. Bernardi, L. Sterpone, M. Violante, and M. Portela-Garcia, "Hybrid fault detection technique: A case study on virtex-II pro's PowerPC 405," IEEE Trans. Nucl. Sci., vol. 53, no. 6, pp. 3550–3557, Dec. 2006.

[Boya13] D. Boyang, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, L. Entrena, "Exploiting the debug interface to support on-line test of Control Flow Errors", 19th IEEE international on-Line testing symposium (IOLTS), 2013.

[Boya14] D. Boyang, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, L. Entrena, "A new solution to on-line detection of Control Flow Errors", 20th IEEE international on-Line testing symposium (IOLTS), 2014.

[Boya15] D.Boyang, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, L. Entrena, "On-line Test of Control Flow Errors: A newDebug Interface-based approach" IEEE Transaction on Computers, 2015.

[Chen05] Y. Chen, "Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring ", IEEE transactions on Computers, vol. 54, no.10, October 2005.

[Chey00] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," IEEE Trans. Nucl. Sci., vol. 47, no. 6, pp. 2231–2236, Dec. 2000.

[DJLu80] D. J. Lu, "Watchdog processors and VLSI" in Proc. Nat. Electron. Conf., vol. 34, 1980, pp. 240–245.

[DJLu82] D. J. Lu, "Watchdog processor and structural integrity checking", IEEE Trans. Computers, vol. C-31, pp. 681–685, July 1982.

[Echt90] K. Echtle, B. Hinz, T. Nikolov, "On Hardware Fault Detection by Diverse Software", Proceedings of the 13-th International Conference on Fault-Tolerant Systems and Diagnostics, 1990, pp. 362-367.

[Eifer84] J. B. Eifert and J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams" in Digest of Papers, 14th Ann. Int. Conf. Fault-Tolerant Computing, 1984, pp. 394–399.

[Enge97] H. Engel, "Data Flow Transformations to Detect Results which are corrupted by hardware faults", Proceedings of the IEEE High-Assurance System Engineering Workshop, 1997, pp. 279-285.

[Entr09] L. Entrena, M. García-Valderas, R. Fernández-Cardenal, M. Portela, C. López-Ongil. "SET Emulation Considering Electrical Masking Effects". IEEE Transactions on Nuclear Science, vol. 56, no. 4, pp. 2021-2025, Aug. 2009.

[Entr12] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela Garcia, C. Lopez-Ongil, "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection," IEEE Transactions on Computers, pp. 313-322, March, 2012.

[Gais01] J. Gaisler "The Leon processor user's manual" Gaisler Research, 2001.

[Gais06] J. Gaisler, M. Isomäki: LEON3 GR-XC3S-1500 Template Design Based on GRLIB, Gaisler Research, Octubre 2006.

[Gais10] J. Gaisler, S. Habinc: GRLIB IP Library User's Manual, Versión 1,0,22, Aeroflex Gaisler, 2010.

[Gall09] M. Gallardo-Campos, M. Portela-Garcia, M. García-Valderas, C. López-Ongil, L. Entrena, M. Grosso, M. Sonza Reorda. "Enhanced Observability in Microprocessor-based Systems for Permanent and Transient Fault Resilience". Proc. 25th Conf. on Design of Circuits and Integrated Systems (DCIS), 2009.

[Golo03] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. "Soft-error detection using control flow assertions". In 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pages 581–588, 2003.

[Gros10] M. Grosso, M. Sonza Reorda, M. Portela-Garcia, M. Garcia-Valderas,C. Lopez-Ongil, L. Entrena "An on-line fault detection technique based on embedded debug features", Proceedings of the 16th IEEE On-LineTesting Symposium, 2010, pp. 167-172.

[Henn96] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", 2nd ed., 1996.

[Hille00] M. Hiller, "Executable assertions for detecting data errors in embedded control systems", Proceedings of the IEEE International Conference on Dependable Systems and Networks, 2000, pp. 24-33.

[Huan99] J. Huang, D.J. Lilja, "Exploiting basic block value locality with block reuse", Proc. Fifth International Symposium on High-Performance Computer Architecture, 1999, pp: 106 – 114.

[Jiri06] Jiri Gaisler, Marko Isomäki: LEON3 GR-XC3S-1500 Template Design Based on GRLIB. Gaisler Research. Octubre 2006.

[Jiri10] Jiri Gaisler, Sandi Habinc: GRLIB IP Library User's Manual. Versión 1.0.22. Aeroflex Gaisler. 2010.

[Joch02] M. Jochim, "Detecting processor hardware faults by means of automatically generated virtual duplex systems", Proceedings of the International Conference on Dependable Systems and Networks, 2002, pp. 399 – 408.

[Kana96] K. Kanawati, N. Krishnamurthy, S. Nair, and J.A. Abraham, "Evaluation of Integrated System-Level Checks for On-Line Error Detection", Proc. IEEE Int'l Symp. Parallel and Distributed Systems, Sept. 1996.

[Made91] H. Madeira, J. Silva, "On-Line testing: Learning and Checking", 2nd International Working Conf. on Dependable comp. for Critical Applications, Tucson, 18-20 February 1991, pp. 170-177.

[Made92] H. Madeira and J. G. Silvia, "On-line signature learning and checking in Dependable Computing for Critical Applications", J. F. Schlichting and R. D. Schlichting, Eds: Springer-Verlag, 1992, pp. 395–420.

[Mahm85] A. Mahmood and E. J. McCluskey, "Watchdog processor: Error coverage and overhead" in Digest, 15th Ann. Int'l. Symp. Fault-Tolerant Computing (FTCS-15), 1985, pp. 214–219.

[Mcfe95] L. Mcfearin and V.S.S. Nair, "Control-Flow Checking Using Assertions", Proc. IFIP Int'l Working Conf. Dependable Computing for Critical Applications, Sept. 1995.

[Mich91] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking Without Program Modification", Proc. 21th Int. Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[Mukh03] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in 36th International Symposium on Microarchitecture, 2003, Proceedings Paper, pp. 29–40, San Diego, CA, USA, Dec 03-05, 2003.

[Nair96] V.S.S. Nair, H. Kim, N. Krishnamurthy, and J.A. Abraham, "Design and Evaluation of Automated High-Level Checks for Signal Processing Applications", Proc. SPIE Advanced Algorithms and Architectures for Signal Processing Conf., Aug. 1996.

[Namj82] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation" in Digest of Papers, IEEE Test Conf., 1982, pp. 461–468.

[Nico03] B. Nicolescu, R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results" Design, Automation and Test in Europe Conference and Exhibition, 2003, pp: 57 – 62.

[Nico04] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip

faults," IEEE Trans. Nucl. Sci., vol. 51, no. 6, pp. 3510–3518, Dec. 2004.

[Nico11] M. Nicolaidis, "Soft errors in modern electronic systems", Springer, 2011.

[NOh02a] N. Oh, P.P. Shirvani, E.J. McCluskey, "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability, Vol. 51, No. 2, 2002, pp. 111-112.

[NOh02b] N. Oh and E. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," IEEE Trans. Reliabil., vol. 51, no. 4, pp. 392–402, Dec. 2002.

[NOh02c] N. Oh, P.P. Shirvani, E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-scalar Processors", IEEE Transactions on Reliability, Vol. 51, No. 1, March 2002, pp. 63-75.

[NOh02d] N. Oh, S. Mitra, E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, Vol. 51, No. 2, February 2002, pp. 180-199.

[Ohls92] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog. In 22nd Int. Symposium on Fault Tolerant Computing, pages 316–325, 1992.

[OhSh02] N. Oh, P. P. Shirvani, E.J. McCluskey, "Control Flow Checking by Software Signature", IEEE transactions on Reliability, vol. 51, no. 2, March 2002, pp. 111-122.

[Parr11] L. Parra, A. Lindoso, M. Portela, L. Entrena, M. Grosso, M. Sonza Reorda, "Control Flow Checking through Embedded Debug Interface", 26thConference on Design of Circuits and Integrated Systems (DCIS), pp. 339-343, 2011.

[Parr13] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, "Efficient Mitigation of Data and Control Flow Errors in Microprocessors", Radiation Effects on Components and Systems (RADECS), 2013.

[Parr14a] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, "Efficient Mitigation of Data and Control Flow Errors in Microprocessors", IEEE Transactions on Nuclear Science (TNS), vol 61, no.4, pp. 1590-1596, 2014.

[Parr14b] L. Parra, A. Lindoso, M. Portela, L. Entrena, D. Boyang, M. Sonza Reorda, L. Sterpone, "A new Hybrid Nonintrusive Error-Detection Technique Using Dual Contro-Flow Monitoring" IEEE Transactions on Nuclear Science (TNS), vol 61, no.6, pp. 3236-3243 , 2014.

[Parr14c] L. Parra, A. Lindoso, L. Entrena, "Comparative of software-based hardening techniques for LEON3 microprocessor" 29thConference on Design of Circuits and Integrated Systems (DCIS), 2014.

[Prat06] B. Pratt, M. Caffrey, P. Graham, E. Johnson, K. Morgan, M. Wirthlin, "Improving FPGA design robustness with partial TMR" Proc. IEEE Int. Rel. Phys. Symp. (IRPS), pp. 27–30, 2006.

[Port12] M.Portela, "Técnicas de Inyección de fallos basadas en FPGAs para la evaluación de la tolerancia a fallos de tipo SEU en circuitos digitales". Leganés, 2007.

[Port12] M. Portela-Garcia,M.Grosso, M. Gallardo-Campos,M. Sonza Reorda, L. Entrena, M. Garcia-Valderas, and C. Lopez-Ongil, "On the use of embedded debug features for permanent and transient fault resilience in microprocessors," Microprocessors Microsyst. , vol. 36, no. 5, pp. 334–343, Jul. 2012.

[Poup05] A. L. Pouponnot "Strategic of SEE Mitigation Techniques for the Development of the ESA Microprocessors: Past, Present and Future", 11th IEEE international On-line Testing Symposium, 2005.

[Prad96] D. K. Pradhan "Fault-Tolerant Computer System Design" Prentice Hall, 1996.

[Reba01] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "A source-to-source compiler for generating dependable software", Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation, 2001, pp. 33-42.

[Reba99] M. Rebaudengo, M. S. Reorda, M. Torchiano, M. Violante, "Soft-error detection through software fault-tolerance techniques", International Symposium on Defect and Fault Tolerance in VLSI Systems, 1999, pp: 210 – 218.

[Rein00] S. K. Reinhardt, S.S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," Proceedings of the 27th

International Symposium on Computer Architecture, 2000, pp. 25-36.

[Reis05] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In Proceedings of the 3rd International Symposium on Code Generation and Optimization, March 2005.

[Reis07] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," IEEE Micro., vol. 27, no. 1, pp. 36–47, Jan. 2007.

[Rodr11] J. Rodrigo, A. Lapolli, L. Rosa and F. Lima "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique". IEEE Transaction on nuclear science, vol. 58, No. 3, June 2011.

[Rodr12] "A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware". IEEE Transaction on nuclear science, vol. 59, No. 4, August 2012.

[Rodr13] J. Rodrigo, M. Altieri, J. Becker and F. Lima "HETA: Hybrid Error-Detection Technique Using Assertions" IEEE Transaction on nuclear science, vol. 60, No. 4, August 2013.

[Rote99] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors", 29-th International Symposium on Fault-Tolerant Computing, 1999, pp. 84-91.

[SAED90] SAED 90 nm Generic Library Synopsys Armenia Educational Department [Online]. Available: http://www.synopsys.com/ Community/UniversityProgram.

[Sanc16] A. Sanchez-Clemente, L. Entrena, and M. Garcia-Valderas, "Partial TMR in FPGAs using approximate logic circuits," IEEE Transactions on Nuclear Science, vol. 63, no. 4, pp. 2233–2240, 2016.

[Shen82] J. P. Shen and M. A. Schuette, "On-line self-monitoring using signatured instruction streams" in Int. Test Conf. Proc., 1982, pp. 275–282.

[Sonz09] M. Sonza. "Software-level soft-error mitigation techniques". 2009.

[Sonz11] M. Sonza Reorda. "Software-level soft-error mitigation techniques". In "Soft errors in modern electronic systems", M. Nicolaidis (Ed.), Springer, 2011.

[SPAR91] "The SPARC architecture manual" version 8, SPARC International, 1991.

[Vemu06] R. Vemu, J.A. Abraham, "CEDA: Control-Flow Error Detection through Assertions", Proc. 12th IEEE International On-Line Testing Symposium (IOLTS), pp. 151-158, 2006.

[Vemu07] R. Vemu, S. Gurumurthy, J.A. Abraham, "ACCE: Automatic Correction of Control-flow Errors", Proc. IEEE Int. Test Conf., 2007, paper 27.2

[Venk03] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Lowcost on-line fault detection using control flow assertions. In International On-Line Testing Symposium, pages 137 – 143, 2003.

[Vint01] J. Vinter, J. Aidemark, P. Folkesson, J. Karlsson, "Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery", Proceedings of the IEEE International Conference on Dependable Systems and Networks, 2001, pp. 347-356.

[Wilk88] K. Wilken, J.P.Shen, "Continuous signature monitoring: efficient concurrent-detection of processor control errors ", International TestConference (ITC), 1988, pp. 914-925.

[Wilk89] K. Wilken and J. P. Shen, "Concurrent error detection using signature monitoring and encryption: Low-cost concurrent-detection of processor control errors" in Dependable Computing for Critical Applications, A. Avizienis and J. C. Laprie, Eds: Springer-Verlag, 1989, vol. 4, pp. 365–384.

[Wilk90] K. Wilken, J. Shen, "Continous Signature monitoring: Low-Cost Concurrent Detection of Processor Control Errors", IEEE Transactions on Computer Aided Design, Vol. 9, No. 6, 1990, pp. 629-641.

[Yau80] S.S. Yau, F.-C. Chen, "An Approach to Concurrent Control Flow Checking", IEEE Transactions on Software Engineering, Vol. 6, No. 2, March 1980, pp. 126-137.

[Yung05] Yung-Yuang Chen, "Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring", IEEE Transactions on Computers, Vol. 54, No. 10, 2005, pp. 1298-1313.