Evaluación de la fiabilidad de microprocesadores COTS mediante las infraestructuras de depuración On-Chip
Dependability Evaluation of COTS Microprocessors via On-Chip debugging facilities

José Isaza-González [1], Alejandro Serrano-Cases[2], Felipe Restrepo-Calle [3], Sergio Cuenca-Asensi[4], Antonio Martínez-Álvarez[5]*
[1,2,4,5] Departamento de Tecnología Informática y Computación, Universidad de Alicante, [1]Centro Regional de Coclé, Universidad Tecnológica de Panamá, [3]Departamento de Ingeniería de Sistemas e Industrial, Universidad Nacional de Colombia
[1]jisaza@dtic.ua.es, [2]aserrano1@dtic.ua.es, [3]ferestrepoca@unal.edu.co, [4]sergio@dtic.ua.es, [5]amartinez@dtic.ua.es

Resumen– *Este artículo presenta una herramienta de inyección de fallos y la metodología para la realización de campañas de inyección de Single-Event-Upsets (SEUs) en microprocesadores Commercial-off-the-shelf (COTS). Este método utiliza las ventajas que ofrecen las infraestructuras de depuración de los microprocesadores actuales, además del depurador estándar de GNU (GDB) para la ejecución y depuración de los programas de pruebas. Los experimentos desarrollados sobre microprocesadores reales, así como en las máquinas virtuales, demuestran la viabilidad y la flexibilidad de la propuesta como una solución de bajo costo para evaluar la fiabilidad de los microprocesadores COTS.*

Palabras claves– *Commercial-off-the-shelf (COTS), depuración integrada en el chip, efectos de la radiación, fiabilidad de microprocesadores, inyección de fallos, errores lógicos.*

Abstract–*This paper presents a fault injection tool and methodology for performing Single-Event-Upsets (SEUs) injection campaigns on Commercial-off-the-shelf (COTS) microprocessors. This method takes advantage of the debug facilities of modern microprocessors along with standard GNU Debugger (GDB) for executing and debugging benchmarks. The developed experiments on real boards, as well as on virtual machines, demonstrate the feasibility and flexibility of the proposal as a low-cost solution for assessing the reliability of COTS microprocessors.*

Keywords– *Commercial-off-the-shelf (COTS), on-chip debug, radiation effects, microprocessors reliability, fault injection, soft-error.*

Tipo de Artículo: Original
Fecha de Recepción: 26 de agosto de 2016
Fecha de Aceptación: 12 de abril de 2017

1. Introducción

En los últimos años, el uso de los dispositivos *comercial-off-the-shelf* (COTS) representa una atractiva alternativa a los componentes diseñados específicamente para trabajar en ambientes de radiación (conocidos como *radiation-hard* o *rad-hard)*. Mediante el uso de dispositivos COTS, es posible reducir significativamente los costes y el tiempo de desarrollo de los sistemas en varios órdenes de magnitud. Además, hay una alta disponibilidad de estos dispositivos en el mercado que ofrecen menor consumo de energía y mayor rendimiento que sus contrapartes *rad-hard* [1].

Sin embargo, la miniaturización de los componentes electrónicos ha provocado consecuencias adversas en la fiabilidad de los sistemas basados en procesadores COTS [2], [3]. En estos sistemas, el efecto de la radiación puede causar fallos de ejecución conocidos como *soft-errors*. Entre ellos, los *Single-Event-Upsets* (SEUs) o alteraciones de evento único [4]. Estos no producen un daño permanente, pero pueden causar un mal funcionamiento de un circuito o un fallo del sistema, que puede ser catastrófico. Un SEU se produce por el impacto de una partícula en el silicio que genera una transición indeseada en el estado de un transistor y

pudiendo provocar el cambio de la información almacenada en una celda de memoria o en un *flip-flop*.

La presencia de SEUs es cada vez más frecuente en los circuitos electrónicos, y por lo tanto es una preocupación creciente en varios dominios de aplicación. Los SEU no solo afectan a los sistemas que operan en entornos hostiles de radiación como los aeroespaciales, sino también a sistemas que operan a bajas altitudes incluso a nivel del mar como los sistemas militares, de automoción, entre otros [5].

La inyección de fallos *hardware* es una técnica que ha sido ampliamente utilizada para evaluar la fiabilidad de estos sistemas en presencia de fallos de esta naturaleza. Los métodos de inyección de fallos *hardware* se pueden clasificar en tres categorías: inyección de fallos físicos, inyección de fallos lógicos mediante la emulación o simulación de circuitos, y la inyección de fallos lógicos mediante las infraestructuras de depuración de procesadores [1].

La inyección de fallos físicos utiliza haces de radiación o de luz láser para inducir SEUs en los circuitos integrados. Este método tiene la ventaja de causar fallos *hardware* reales en los sistemas. Sin embargo, estos experimentos son muy caros y requieren instalaciones especiales [6].

La inyección de fallos lógicos por emulación o simulación de circuitos se realiza utilizando plataformas de emulación de *hardware*, y lenguajes de descripción *hardware* (*Hardware Description Languages* o HDLs), respectivamente [7]. La inyección de fallos se implementa mediante el modelo de fallo *bit-flip*, en el que el contenido de una celda de memoria se invierte en el momento de la inyección. Comúnmente, los circuitos que se pretende evaluar (*Device Under Tests* o DUTs) se modelan mediante lenguajes HDLs y se prototipan en uno o más dispositivos de lógica programable (*Field Programable Gate Arrays o* FPGAs). Para inyectar fallos sobre el prototipo *hardware* es necesario construir recursos lógicos específicos dentro del circuito con fines de depuración, lo que permite reunir información sobre el funcionamiento del circuito en presencia de fallos, y evaluar el comportamiento de los mecanismos de tolerancia a fallos [8]–[10]. Otro enfoque lleva a cabo la inyección de fallos mediante el uso de las capacidades de reconfiguración de las FPGAs [11]. Estas técnicas tienen severas limitaciones en la evaluación de la fiabilidad de procesadores COTS, ya que no es común que los fabricantes publiquen modelos o descripciones RTL detalladas de sus procesadores.

La tercera técnica se basa en la utilización de las infraestructuras de depuración integradas en los propios procesadores, para realizar la inyección de fallos y observar sus efectos. Estas técnicas no requieren, por tanto, el desarrollo de bloques *hardware* específicos adicionales como en el caso anterior. Las estructuras de depuración integradas están destinadas originalmente a otros fines. Un ejemplo son las cadenas *test Boundary Scan* o las infraestructuras de depuración integradas (*On-Chip Debugging* u OCD).

En concreto, las infraestructuras OCD permiten depurar el código que se ejecuta en el procesador desde un computador externo. Este computador utiliza una herramienta *software* o depurador que se conecta al OCD mediante un puerto específico (generalmente serie). Este *software* permite controlar la ejecución y al mismo tiempo observar el estado de los elementos internos del procesador (memoria, registros, puertos, etc.) en cada momento [12], [13]. Con esta aproximación se pueden emular fallos en dispositivos reales (no meros prototipos o modelos), sin embargo, su aplicación está limitada por las capacidades de OCD, e implica el diseño de una configuración experimental, incluyendo *hardware* y *software*, para cada sistema.

La metodología que se propone en este trabajo, está basada en una nueva herramienta de bajo coste para inyección de fallos lógicos, concebido para evaluar la tolerancia a SEUs de procesadores COTS. Esta herramienta no requiere el diseño adicional de módulos de *hardware* externo o interno, solo hace uso de un recurso común integrado (temporizador) para acelerar la inyección de fallos. Adicionalmente, utiliza las infraestructuras de depuración *hardware* tales como OCD, y el depurador estándar de GNU (GDB). La principal ventaja de este método es la alta portabilidad a diferentes arquitecturas de procesadores y plataformas de emulación / simulación. Por otra parte, nuestra propuesta es compatible con cualquier tipo de procesador (*softcore* y *hardcore*) y específicamente dispositivos COTS.

Este trabajo está organizado de la siguiente manera: en la sección 2 se resumen los trabajos anteriores con respecto a los métodos de inyección de fallos en componentes COTS; la sección 3 detalla la herramienta y la metodología propuesta; en la sección 4 se analizan

los resultados experimentales obtenidos en la evaluación de diferentes microprocesadores y extiende nuestra propuesta presentada en [14] que además de inyección de fallos sobre el banco de registros del procesador se incluyen los resultados obtenidos sobre la memoria RAM (*Random Access Memory*); finalmente, la sección 5 presenta las conclusiones del trabajo y propone algunas mejoras en el futuro.

2. Trabajos previos

Se han propuesto varios trabajos para evaluar la tolerancia a fallos y la fiabilidad de los microprocesadores COTS en presencia de fallos tipo SEUs. Por un lado, las técnicas *software* presentan un método de inyección de fallos mediante la ejecución de instrucciones añadidas al programa. Aunque, el tiempo necesario para inyectar el fallo es relativamente corto, estos métodos son muy intrusivos ya que requieren la modificación del código original. En [15] Velazco et al. describen una técnica *software* basada en rutinas de interrupción de servicio denominada CEU (*Code Emulated Upset*) para inyección de fallos. La implementación del método se realiza sobre una arquitectura *hardware*, denominada THESIC *(Testbed for Harsh Environment Studies on Integrated Circuits*), es una plataforma genérica que se encarga de generar las interrupciones que activan la inyección, las localizaciones del fallo, y de leer los resultados. Este enfoque necesita *hardware* adicional para activar los procedimientos de interrupción, automatizar la carga de memoria con los datos correspondientes al código CEU deseada, comparar los resultados con los resultados esperados y controlar el tiempo de ejecución.

Por otro lado, las técnicas basadas en las infraestructuras de depuración integradas en el procesador no requieren la modificación del código de aplicación. Desde un punto de vista general, OCD puede ser definida como la combinación integrada de *hardware* y *software*, que permite acceder a los recursos internos durante la ejecución del sistema para la inyección de fallos [16], [17]. La implementación de las infraestructuras de depuración varía de acuerdo a las diferentes familias de procesadores, por lo tanto también sus características y capacidades. El acceso a los recursos internos (registros, memoria, etc.) se realiza generalmente a través de la interfaz estándar JTAG (*Joint Test Action Group*).

La arquitectura genérica de un procesador provisto de una infraestructura OCD se compone principalmente de un dispositivo bajo prueba (*Device Under Test o DUT*) y un controlador externo, que puede ser un ordenador central (*host*) o un componente *hardware* asociado para realizar campañas de inyección de fallos.

La comunicación entre el controlador externo y el DUT también se realiza a través de la interfaz JTAG.

Basado en este concepto, el trabajo presentado en [16] propone una metodología escalable. Los autores modifican interfaces *hardware* y circuitos OCD dedicados a ayudar a la ejecución de las campañas de inyección de fallos en tiempo real. El objetivo principal de este enfoque es permitir la inyección de fallos en elementos de memoria del microprocesador.

Un enfoque similar se presenta en [18], donde un módulo *hardware* específico se ejecuta como interfaz entre el DUT y el *host*. La técnica se basa en el estándar de depuración Nexus. Este estándar proporciona una interfaz de propósito general para la depuración en procesadores empotrados y el desarrollo de herramientas de depuración. En este caso, los fallos se inyectan sin alterar su ejecución. La aplicabilidad de esta propuesta está condicionada a la compatibilidad del microprocesador con el estándar Nexus. Además, se presentó en [19] una técnica de instrumentación de circuito para medir la sensibilidad SEU en los procesadores complejos. Este método emplea un módulo *hardware* a medida, y una unidad de control conectada al OCD a través de la interfaz JTAG. La unidad de control lleva a cabo la inyección de fallos, y controla las funcionalidades del OCD. Esta técnica no requiere la modificación del sistema destino; no es intrusiva, pero requiere el diseño de una unidad de control específica.

También existen otras propuestas para inyectar fallos a través de GDB [20], [21]. Sin embargo, el proceso para generar secuencias de comandos estáticos y la necesidad de ejecutar el depurador para cada inyección hace que los métodos sean lentos. En [20] se presenta una herramienta de inyección SFIG (*Software-based Fault Injection using gdb*), donde GDB inyecta el fallo cuando el contador de programa llega a la dirección de la instrucción dada. Si esa dirección está dentro de un bucle, GDB es el encargado de controlar el número de iteraciones, haciendo el proceso más ineficiente. En [21] la herramienta de inyección FAUST (*FAUlt-injection*

Script-based Tool) utiliza temporizadores de *software* para seleccionar el tiempo de inyección, haciendo que el proceso sea más impreciso. Estos enfoques están concebidos para ser aplicados a los procesadores que se ejecutan sobre el sistema operativo Unix.

Nuestro enfoque es compatible para los procesadores que se ejecutan con o sin sistema operativo. Además, incluye algunas mejoras para acelerar la inyección de fallos y el proceso de selección exacto del ciclo de inyección. Esto permite un análisis a posteriori del fallo, si es necesario. Además, el concepto de tiempo crítico se incluye para apoyar un control de grano fino del tiempo de ejecución. Este concepto es importante cuando algún tipo de técnica de tolerancia a fallos basada en *software* está incluido en el código.

3. Sistema de inyección propuesto

El sistema de inyección de fallos propuesto realiza la inyección de SEUs y observación de sus efectos, mediante pruebas experimentales llevadas a cabo directamente en el DUT. Nuestra propuesta utiliza la infraestructura estándar de depuración GDB para las pruebas experimentales de inyección de fallos, que inciden en el contenido de la memoria RAM y los registros (incluyendo registros de propósito general y registros especiales). Además, utiliza un módulo *hardware*, temporizador, que es común en los procesadores COTS actuales.

Con el fin de demostrar la viabilidad de este método se ha desarrollado la herramienta de inyección de fallos, que se puede aplicar a procesadores reales, y también es capaz de trabajar en diferentes ambientes tales como simuladores, emuladores y máquinas virtuales.

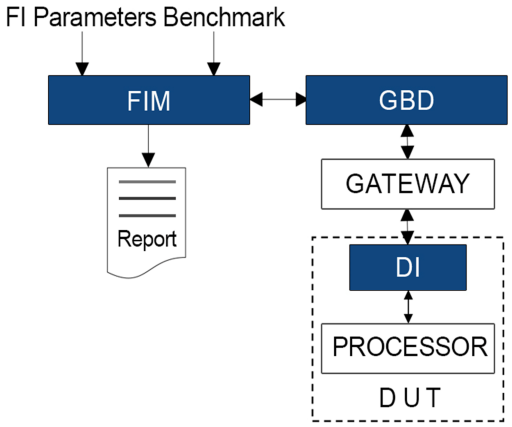

Figura 1. Sistema de inyección de fallos propuesto.

Nuestra herramienta cuenta con cuatro componentes principales (figura 1): el gestor de inyección de fallos (*Fault Injection Manager* o FIM), el depurador de GNU (GDB), la puerta de enlace (Gateway) y la interfaz de depuración (*Debugging Interface* o DI). FIM está a cargo de la gestión y generación de campañas de inyección de fallos y analizar y clasificar los resultados. Además, FIM genera los *scripts* de inyección de fallos correspondientes para GDB con los parámetros de la campaña proporcionados por el usuario. GDB es el depurador utilizado para realizar la inyección de fallos en el DUT. La puerta de enlace es específica para cada DUT y permite la comunicación entre GDB y el DUT. En el caso de procesadores COTS, la puerta de enlace es puesta en marcha por el *host* para establecer la comunicación; de lo contrario, la puerta de enlace está integrada en el emulador / simulador. La DI implementa los comandos de GDB, tales como la descarga de código, la ejecución paso a paso a través del código, o la inserción de puntos de ruptura. Esto varía según el DUT, en el caso de procesadores COTS la DI se lleva a cabo mediante el OCD, mientras que para los emuladores / simulador se utiliza una versión de software conocido como GDB *stub*. La campaña de inyección de fallos es administrado por el *host* en donde la FIM se está ejecutando, por lo que en ambos casos la velocidad del proceso de inyección está limitado por la comunicación necesaria entre el *host* y el DUT.

3.1 Fases de inyección de fallos

La campaña de inyección está compuesta por dos fases principales: la fase de inicialización y la fase de inyección en tiempo de ejecución.

La fase de inicialización incluye todas las tareas realizadas por el FIM para preparar una campaña de inyección de fallos para una aplicación específica (*Benchmark*).

En primer lugar el FIM ejecuta el programa sin realizar inyección de fallos para obtener una ejecución de referencia almacenando el resultado correcto de la aplicación objetivo (ejecución de oro o *golden*); también, recopila información valiosa de la ejecución, como la dirección inicial y final de memoria del programa y el número total de pasos de ejecución. Una vez recopilada esta información, el FIM genera automáticamente los *scripts* de inyección con la información de configuración acerca de cuándo y dónde

se debe inyectar un fallo, el número total de fallos por campaña y las condiciones de finalización como lo es el tiempo de espera (*timeout*) utilizado para controlar la terminación anormal del programa.

En segundo lugar, en los casos en donde el DUT es un procesador COTS, el código original tiene que ser ligeramente instrumentado para acelerar el proceso de inyección. De hecho, la forma habitual para llegar hasta el punto de ejecución, donde se va a inyectar un fallo es mediante la ejecución lenta del comando *stepi* en GDB.

Para evitar este cuello de botella, proponemos el uso de un temporizador *hardware* (*timer*) para interrumpir la ejecución del sistema en el ciclo seleccionado y permitir que GDB realice la inyección del fallo de forma rápida. El temporizador se ejecuta paralelamente al procesador y su rutina de interrupción de servicio (*Interrupt Service Routine* o ISR) lo que genera gastos generales (*overhead*) insignificantes de código y tiempo de ejecución (alrededor de 40 ciclos de reloj).

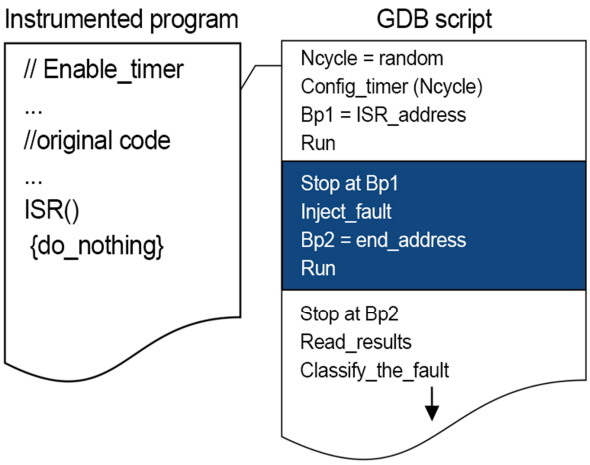

Figura 2. Metodología de inyección de fallos en procesadores COTS.

La figura 2 muestra el procedimiento. Como se puede observar, se realiza una instrumentación del código fuente añadiendo líneas de códigos necesarias para configurar y habilitar el temporizador. El temporizador y la ISR solo se utilizarán para interrumpir la ejecución del sistema en un determinado número de ciclos de reloj, en donde GDB inserta un punto de ruptura que se utiliza para iniciar la rutina de inyección de GDB.

La fase de inyección en tiempo de ejecución realiza la inyección de fallos y la clasificación de cada fallo. El FIM inicia una sesión de depuración y, al azar selecciona un ciclo de reloj y un bit dentro de los recursos internos del procesador. Luego, continúa la ejecución del programa hasta que se interrumpe por la ISR mediante un contador de ciclos en el caso de los procesadores COTS, o al adelantar el número de pasos de ejecución establecidos en el caso del simulador / emulador. Después de la inyección de un fallo, la ejecución se reanuda hasta el final del programa.

3.2 Método de Inyección de fallos y Clasificación

El método de inyección de fallo se basa en la inyección de un SEU (*bit-flip*) en cada ejecución del programa, que se realiza al azar en tiempo y ubicación, afectando los datos almacenados en los recursos internos del procesador (tal como la memoria o banco de registros). El procedimiento para realizar un *bit-flip* es el siguiente: En primer lugar, se lee la ubicación y el contenido del elemento de memoria en el que se inyecta el fallo. En segundo lugar, se realiza la operación XOR al valor en la ubicación establecida con una máscara de señalización, que selecciona de forma aleatoria el bit interno del procesador que se va a cambiar durante un ciclo de reloj. Por último, se escribe de nuevo el valor después de que se da el cambio a la ubicación original.

Los efectos de los fallos se clasifican de acuerdo a las siguientes categorías [22]:

- *unnecessary for Architecturally Correct Execution* (unACE): en caso de que el programa completa su ejecución y obtiene los resultados esperados.
- *Silent Data Corruption* (SDC): cuando el fallo no se ha detectado o corregido y hace que el programa complete su ejecución normalmente pero no obtiene los resultados esperados.
- *Hang* o *Bloqueado*: cuando el fallo causa una terminación anormal del programa o un bucle infinito.

4. Resultados Experimentales

Para evaluar la metodología de inyección de fallos, se exploraron dos casos de estudios basados en diferentes DUT. Para el primero, el DUT es un procesador *Texas Instruments MSP430*, mientras que el segundo es un procesador ARMV4T simulado mediante QEMU.

Las principales características del procesador MSP430 son: 16-bit RISC (*Reduced Instruction Set*

Computer) CPU, 16 KB de memoria flash, y 512 bytes de memoria RAM. El procesador tiene un banco de registros con 16 registros (R0-R15). Los primeros cuatro registros están destinadas para fines especiales. R0 se reserva para el contador de programa (*Program Counter* o PC), R1 es el puntero de pila (*Stack Pointer* o SP), y R2 es el registro de estado (*Status Register* o SR).

Este registro contiene el estado de la CPU MSP430, que se define por un conjunto de bits. El Registro de Estado almacena el contenido de los indicadores aritméticos (*Carry, Overflow, Negative*, y *Zero*), así como algunos bits de control tales como SCG1 ((*System Clock Generator 1*)), SCG0 (*System Clock Generator 0*), OSCOFF (*Oscillator Off*), CPUOFF (*CPU Off*) que se utiliza para controlar el modo de funcionamiento de la CPU. El bit general de habilitación de interrupción (*General Interrupt Enable* o GIE) se utiliza para habilitar o deshabilitar las interrupciones enmascarables.

El registro R3 se utiliza para la generación constante. Los registros restantes desde R4 a R15 son registros de propósito general.

QEMU es un emulador y virtualizador genérico, es decir, que es capaz de simular un sistema completo. Además, tiene la ventaja de ser capaz de funcionar tanto como un emulador puro o como una máquina virtual nativa (en x86 / arquitectura x86-64). Es posible también, utilizarlo para ejecutar y depurar una aplicación de sistema operativo menos independiente.

La arquitectura ARMV4T proporciona 16 registros de propósito general en el modo de usuario, todos los cuales son de 32 bits de ancho. R15 es el contador de programa (*Program Counter* o PC), pero puede ser manipulado como un registro de propósito general. R14 se utiliza como un registro de enlace (*Link Register* o LR) por la instrucción de bifurcación y enlace. R13 se suele utilizar como puntero de pila (*Stack Pointer* o SP).

El registro de estado actual del programa (*Current Program Status Register* o CPSR) contiene cuatro indicadores de condición de 1 bit (*Negative, Zero, Carry*, y *Overflow*) y cuatro campos que reflejan el estado de ejecución del procesador. El resto de los registros de la CPU, desde R0 a R12, son registros de propósito general.

El conjunto de programas de pruebas (*test benchmark*) utilizado en los experimentos se compone de los siguientes programas: Reducción de vector (VR), el algoritmo de ordenamiento Quicksort (QuickSort y QuickSort _crc), y la multiplicación de matrices (MxM y MxM_crc). En los dos últimos *benchmarks*, se ha añadido la comprobación de redundancia cíclica (*Cyclic Redundancy Check* o CRC) para reducir la amplitud de los datos (*datawidth)* de los resultados y para facilitar la comprobación de la exactitud. CRC es una técnica para la detección de errores en los datos digitales, que se utiliza comúnmente en las redes de telecomunicaciones y dispositivos de almacenamiento digital. Además, esta técnica utiliza división polinómica para determinar un valor llamado CRC, que es generalmente la anchura de 16 o 32 bits. Como técnica de reducción, dos valores CRC no coincidirán si se cambia cualquier bit de un conjunto de bits determinado. Utilizamos CRC para evaluar la coherencia de los resultados, en comparación con los programas de pruebas con o sin algoritmo CRC.

4.1 Configuración de campañas de Inyección de fallos

Para evaluar el rendimiento del método de inyección en diferentes partes del DUT (tales como los registros del procesador y la memoria RAM), se midió el tiempo necesario para inyectar un solo fallo durante 1000 pruebas. Los resultados promedios obtenidos fueron 1,10 segundos y 0,12 segundos para MSP430 y QEMU-ARMV4T, respectivamente. El primer resultado revela una gran dependencia de las latencias de comunicación entre el *host* y el DUT en el caso de MSP430.

4.2 Campañas de inyección de fallos en registros de la CPU

Para verificar la coherencia de los resultados, se realizaron dos campañas de inyección de fallos para MSP430 y ARMV4T inyectando 1,000 SEUs en cada registro del banco de registros. Es decir, un total de 16000 SEUs por campaña en MSP430 y 17000 SEUs en ARMV4T. Dos *benchmarks* fueron probados en cada campaña: QuickSort y MxM. La figura 3 y la figura 4 muestran los resultados.

Como era de esperarse, los fallos solo afectan a los registros que están implicados en las instrucciones reales del programa. Por ejemplo, los fallos inyectados sobre los registros que no son utilizados durante la ejecución del programa (por ejemplo, R4 y R5 en QuickSort en MSP430), se clasifican como unACE (figura 3). En ambos *benchmarks* ejecutados sobre MSP430, los registros críticos, tales como R0 (contador de programa) y R1 (puntero de pila) presentan las tasas

más altas de error. Esto es particularmente cierto en el caso de QuickSort para el registro R1/SP, ya que este algoritmo se basa principalmente en la recursión; por lo tanto, utiliza la pila ampliamente, por lo que el registro R1/SP, es el más crítico.

Vale la pena señalar que los porcentajes de fallos clasificados como unACE de los registros desde R6 a R12 son bastante altos debido a que el tiempo de vida de estos registros es relativamente corto, es decir, que no eran ampliamente propensos a fallos durante la ejecución del programa. Por el contrario, otros registros altamente utilizados (tales como R14 y R15 en QuickSort; y de R13 a R15 en el caso de MxM) se utilizan ampliamente para realizar muchos cálculos durante todo el proceso y, en consecuencia, presentan mayor porcentaje de efectos indeseables. En particular, en el caso de MxM, los registros desde R13 a R15 se utilizan para calcular y almacenar los resultados intermedios dentro del programa, por lo tanto, causan e incrementan la tasa de fallos SDC.

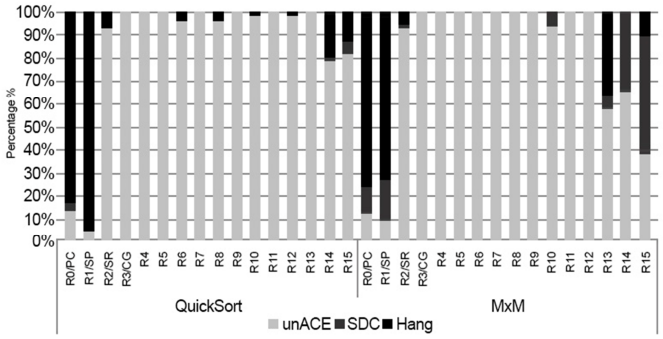

Figura 3. Porcentajes de clasificación de fallos contra el banco de registros del procesador TI-MSP430.

En cuanto a QuickSort en la arquitectura ARMV4T (figura 4), el registro R14/LR presenta un notable incremento en su tasa de fallos de tipo *Hang* en comparación con MxM. Esto es debido al incremento en su vida útil, ya que se utiliza para guardar la dirección de retorno de las subrutinas. QuickSort tiene dos rutinas grandes y recursivas, que aumentan la probabilidad de error.

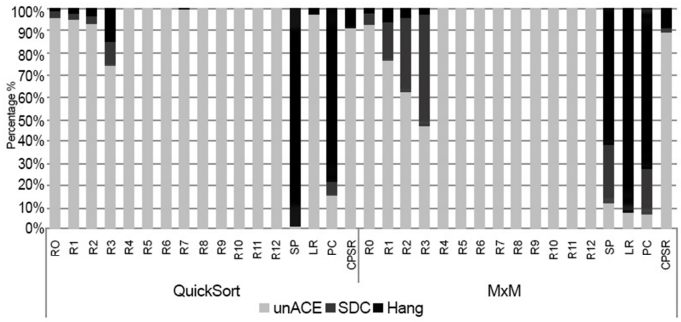

Figura 4. Porcentajes de clasificación de fallos contra el banco de registros del procesador ARMV4T (emulado con Qemu).

En resumen, los resultados globales de inyección en ambas arquitecturas RISC son bastante similares como se esperaba.

La segunda prueba demuestra la fiabilidad global de la aplicación, cuando los fallos se inyectan en todo el banco de registros. Este conjunto de *benchmarks* han sido probados en ambas arquitecturas inyectando 1,000 SEUs por registro.

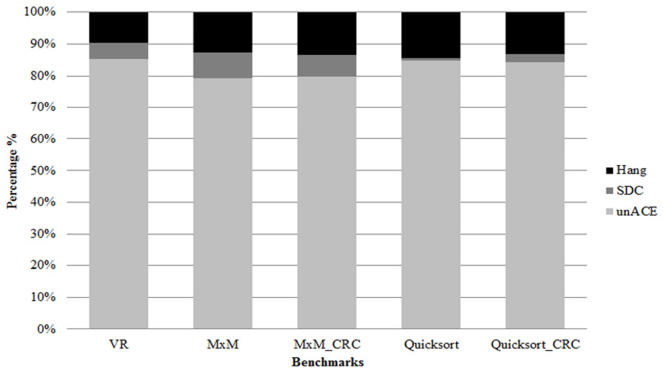

Figura 5. Porcentajes generales de clasificación de fallos para cada *benchmak* ejecutado en el procesador TI-MSP430.

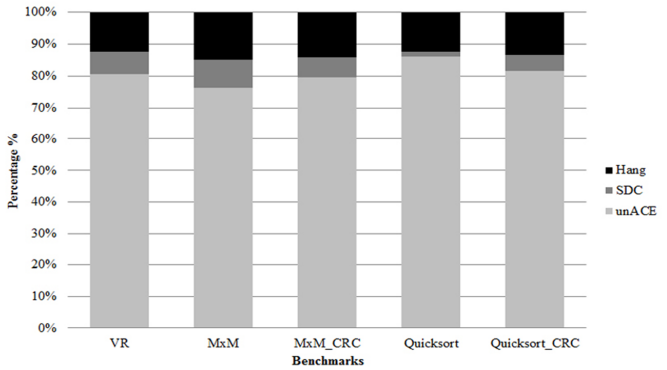

Figura 6. Porcentajes generales de clasificación de fallos para cada *benchmark* ejecutado en el procesador ARMV4T.

En promedio, MSP430 ofrece mejor fiabilidad que ARM, ya que sus porcentajes de fallos unACE son más altos en todos los *benchmarks*. Teniendo en cuenta la distinta naturaleza de los *benchmarks*, varias características pueden ser señaladas a continuación. Estos algoritmos se centraron en el procesamiento de datos, es decir, VR, MxM y MxM_crc son más propensos a ser afectados por fallos que causan SDC.

Por otra parte, los *benchmarks* de QuickSort y QuickSort_crc, que son más orientados a control de flujo, por lo que presentan una vulnerabilidad de menor importancia a fallos en las variables locales. Esta peculiaridad es causada debido al uso intensivo de la pila para guardar las variables automáticas durante el flujo de recursividad (proceso de pila). Respecto a la influencia del cálculo CRC en MxM y QuickSort, los resultados son bastante diferentes. En el caso de la multiplicación de matrices, CRC aumenta ligeramente la fiabilidad del sistema (porcentajes unACE). Por el contrario, para QuickSort, el número de efectos indeseables (SDC + Hang) aumenta cuando se calcula CRC. Este efecto se ve claramente en ARMV4T en el que el porcentaje de SDC para la versión CRC es más del doble del porcentaje de QuickSort.

4.3 Campañas de inyección de fallos en la memoria RAM

Como última prueba, hemos realizado campañas de inyección sobre los recursos de la memoria RAM. Se realizaron dos campañas de inyección de fallos para MSP430G2553 y MSP430F5529 inyectando 16,000 SEUs en cada sección de la memoria RAM, que resume un total de 48,000 SEUs por campaña. Dos *benchmarks* fueron utilizados en cada campaña: QuickSort y MxM.

Los resultados obtenidos en ambos procesadores demuestran que los efectos de los fallos varían de acuerdo a la sección de memoria RAM que se considere para la inyección. La figura 7, muestra los porcentajes de clasificación de los fallos. En el caso de la sección de datos, que tiene un tamaño fijo establecido y contiene todas las variables, que pueden ser utilizados durante el tiempo de ejecución del programa. Por lo tanto, esta sección es más propensa a ser afectada por fallos que causan SDC. Mientras tanto, la sección de pila, además de contener los parámetros de función, almacena sus direcciones de retorno aumentando así el número de *Hangs*. El número de datos / direcciones en la pila varía durante la ejecución del programa y, en general, es menor que las variables críticas en la sección de datos, en nuestro caso se aumenta el porcentaje de fallos unACE.

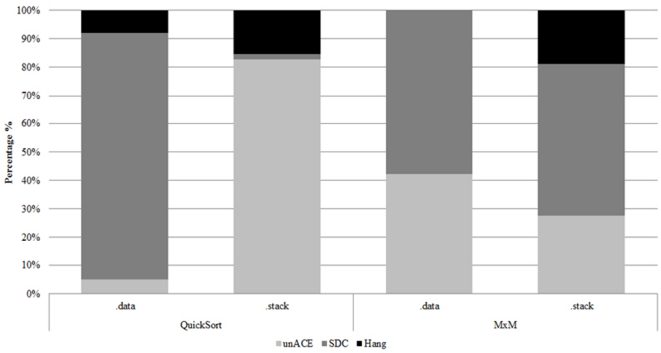

Figura 7. Porcentajes de clasificación de fallos contra las secciones de memoria RAM en el procesador TI-MSP430.

La figura 8 resume la información mediante la recopilación de los fallos obtenidos de todas las campañas de inyección, tanto en el banco de registros como en la memoria RAM. La tasa de error del banco de registros es menor como su porcentaje de fallos un ACE es mayor que el un ACE de la memoria RAM. Los resultados indican que la memoria RAM es más propensa a ser afectado por fallas que causan SDC.

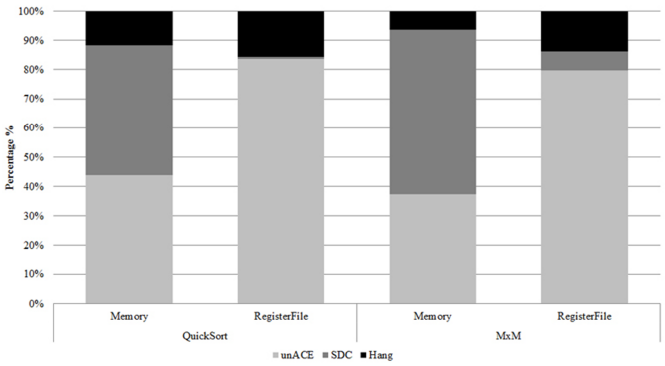

Figura 8. Porcentajes generales de clasificación de fallos conta la memoria RAM y el banco de registros en el procesador TI-MSP430.

En promedio, los resultados globales de los fallos (figura 9), teniendo en cuenta tanto los fallos inyectados en el banco de registros y la memoria RAM, muestran que el 58 % de los fallos inyectados se clasifican como unACE para MxM, y el 63% para quickSort. El porcentaje de efectos indeseables (SDC + Hang) son del 37 % para quickSort y el 42 % para MxM.

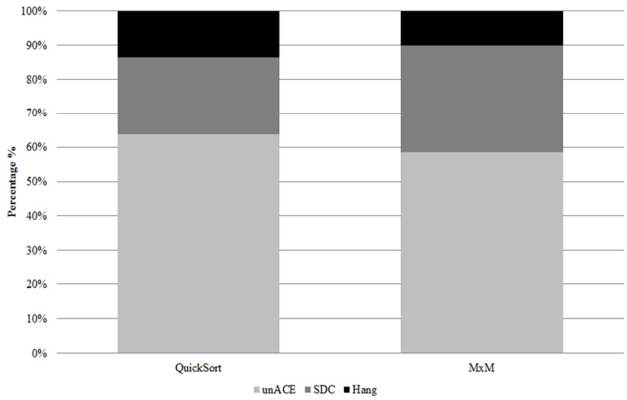

Figura 9. Porcentajes generales de clasificación de fallos contra la memoria RAM y el banco de registros para cada benchmark ejecutado en el procesador TI-MSP430.

5. Conclusiones

El sistema propuesto es una herramienta destinada a la evaluación de la fiabilidad de los microprocesadores COTS. La aplicación automática y el mínimo impacto sobre el funcionamiento del microprocesador son algunas de las principales características de nuestra propuesta. Además, esta herramienta no requiere ningún hardware adicional de propósito específico, y tiene bajos costes de desarrollo e implementación. Las pruebas realizadas muestran la flexibilidad de la herramienta para adaptarse a diferentes plataformas, así como la coherencia de los resultados de la inyección de fallos que afectan a los registros de la CPU y a las secciones de la memoria RAM. Además, el uso de la herramienta con las plataformas de emulación / simulación con soporte para GDB, como también QEMU, permite la aceleración de la campaña de inyección de fallos, ofreciendo estimaciones con respecto a la fiabilidad del sistema.

6. Agradecimientos

Este trabajo fue financiado por el Ministerio español de Economía y Competitividad y el Fondo Europeo de Desarrollo Regional con el proyecto "Evaluación Temprana de los Efectos de radiación Mediante simulación y virtualización. Estrategias de mitigación en arquitecturas de microprocesadores Avanzados "(Ref: ESP2015-68245-C4-3-P MINECO / FEDER, UE), y la Universidad Nacional de Colombia con el proyecto "Desarrollo de *software* Una métrica de Vulnerabilidad de registros en Microprocesadores COTS" (Cod HERMES: 28433).

7. Referencias

[1] M. Nicolaidis, Ed., *Soft Errors in Modern Electronic Systems*, vol. 41. Boston, MA: Springer US, 2011.

[2] R. C. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Trans. Device Mater. Reliab.*, vol. 5, no. 3, pp. 305–315, Sep. 2005.

[3] P. Shivakumar, M. D. Kistler, S. W. Keckler, D. C. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” *Dependable Syst. Networks, 2002. DSN 2002. Proceedings. Int. Conf.*, pp. 389–398, 2002.

[4] T. Karnik, P. Hazucha, and J. Patel, “Characterization of soft errors caused by single event upsets in CMOS processes,” *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 2, pp. 128–143, 2004.

[5] F. Wang and V. D. Agrawal, “Single Event Upset: An Embedded Tutorial,” in *21st International Conference on VLSI Design (VLSID 2008)*, 2008, pp. 429–434.

[6] H. Quinn, P. Graham, J. Krone, M. Caffrey, and S. Rezgui, “Radiation-induced multi-bit upsets in SRAM-based FPGAs,” *Nucl. Sci. IEEE Trans.*, vol. 52, no. 6, pp. 2455–2461, 2005.

[7] H. M. Quinn, D. a. Black, W. H. Robinson, and S. P. Buchner, “Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing,” *IEEE Trans. Nucl. Sci.*, vol. 60, no. 3, pp. 2119–2142, 2013.

[8] P. Folkesson, S. Svensson, and J. Karlsson, “A comparison of simulation based and scan chain implemented fault injection,” in *Digest of Papers. Twenty-Eighth Annual International*

Symposium on Fault-Tolerant Computing (Cat. No.98CB36224), 1998, pp. 284–293.

[9] D. Alexandrescu, L. Sterpone, and C. López-Ongil, "Fault injection and fault tolerance methodologies for assessing device robustness and mitigating against ionizing radiation," *19th IEEE Eur. Test Symp.*, 2014.

[10] R. V. T. Calin, M. Nicolaidis, "Upset hardened memory design for submicron CMOS technology," . *IEEE-Transactions-on-Nuclear-Science*, vol. 43, no. 6, pp. 2874–2878, 1996.

[11] L. Antoni, R. Leveugle, and M. Feher, "Using run-time reconfiguration for fault injection in hardware prototypes," in *17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings.*, 2002, pp. 245–253.

[12] M. Portela-Garcia, C. Lopez-Ongil, M. Garcia Valderas, and L. Entrena, "Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 2, pp. 308–314, Mar. 2011.

[13] L. Parra *et al.*, "Efficient Mitigation of Data and Control Flow Errors in Microprocessors," *IEEE Trans. Nucl. Sci.*, vol. 61, no. 4, pp. 1590–1596, 2014.

[14] J. Isaza-Gonzalez, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martinez-Alvarez, "Dependability evaluation of COTS microprocessors via on-chip debugging facilities," in *2016 17th Latin-American Test Symposium (LATS)*, 2016, pp. 27–32.

[15] R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2405–2411, 2000.

[16] A. V. Fidalgo, M. G. Gericota, G. R. Alves, and J. M. Ferreira, "Real-time fault injection using enhanced on-chip debug infrastructures," *Microprocess. Microsyst.*, vol. 35, no. 4, pp. 441–452, 2011.

[17] S. A. Chekmarev and A. M. F. Reshetnev, "Modification of Fault Injection Method via On-Chip Debugging for Processor Cores of Systems-On-Chip," pp. 1–4, 2015.

[18] Junjie Peng, Jun Ma, Bingrong Hong, and Chengjun Yuan, "Validation of Fault Tolerance Mechanisms of an Onboard System," in *2006 1st International Symposium on Systems and Control in Aerospace and Astronautics*, pp. 1230–1234.

[19] M. Portela-Garcia, C. Lopez-Ongil, M. Garcia-Valderas, and L. Entrena, "A Rapid Fault Injection Approach for Measuring SEU Sensitivity in Complex Processors," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, 2007, pp. 101–106.

[20] Rajesh Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pp. 137–143.

[21] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, I. Solcia, and L. Tagliaferri, "FAUST: fault-injection script-based tool," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, p. 160.

[22] A. Martínez-Álvarez, F. Restrepo-Calle, L. A. Vivas Tejuelo, and S. Cuenca-Asensi, "Fault tolerant embedded systems design by multi-objective optimization," *Expert Syst. Appl.*, vol. 40, no. 17, pp. 6813–6822, 2013.