Mitigación automática de *soft-errors* en nuevas aplicaciones de los sistemas empotrados

Felipe Restrepo-Calle[1], Antonio Martínez-Álvarez[1], H. Guzmán-Miranda[2]
F.R. Palomo[2], M.A. Aguirre[2], Sergio Cuenca-Asensi[1]

[1]Departamento Tecnología Informática y Computación. Universidad de Alicante

[2]Departamento Ingeniería Electrónica. Universidad de Sevilla

Resumen

Las crecientes prestaciones de los microprocesadores, derivadas de la miniaturización de las tecnologías electrónicas, están provocando la aparición de nuevas aplicaciones de los sistemas empotrados en todos los ámbitos. Sin embargo, el empleo de las tecnologías nanométricas conlleva una mayor sensibilidad de los procesadores a los fallos transitorios inducidos por radiación (*soft-errors*). Por tanto, en el desarrollo de las nuevas generaciones de estos sistemas, y no solo en aquellos que deben trabajar en ambientes de alta radiación, la fiabilidad se está convirtiendo en un factor de creciente importancia. En este trabajo se presenta una infraestructura basada en compilador que permite el co-diseño de estos nuevos sistemas donde la tolerancia a fallos es un parámetro tan relevante como lo pueden ser el coste, el consumo o el rendimiento. Las herramientas desarrolladas facilitan la exploración del espacio de diseño existente entre las técnicas de protección puramente hardware y las técnicas basadas en la redundancia del software. De esta forma se obtienen soluciones híbridas con un mejor balance entre los distintos requisitos del diseño, habilitando a los sistemas empotrados para abordar nuevas aplicaciones de seguridad y misión crítica.

1. Introducción

La incorporación de las tecnologías nanométricas en la implementación de procesadores (tanto *soft-cores* como *hard-cores*), ha puesto a disposición de los diseñadores una nueva generación de sistemas empotrados y sistemas *on-chip* de gran capacidad. Estos nuevos sistemas mejoran notablemente las prestaciones de los antiguos, permitiéndo además su aplicación a nuevos ámbitos y mercados donde anteriormente los sistemas empotrados no tenían cabida. Por otro lado, estas nuevas tecnologías además de reducir el tamaño de los componentes, con las consiguientes mejoras en costes y rendimiento, reducen también los márgenes de diseño (niveles de voltaje y ruido soportados). Esto redunda negativamente en la fiabilidad de los dispositivos y provoca que los microprocesadores sean más susceptibles a *fallos transitorios*, como los inducidos por radiación [1]. Este tipo de fallos, también llamados *soft-errors*, no provocan un daño permamente, pero pueden ocasionar un comportamiento anómalo en la ejecución de un programa alterando la transferencia de alguna señal o incluso alguna variable registrada.

Hasta hace poco los fallos inducidos por radiación afectaban principalmente a sistemas cuya operación se llevaba a cabo en entornos espaciales sometidos a radiación. Sin embargo, estas nuevas circunstancias están provocando la aparición de *soft-errors* también en sistemas que trabajan a bajo nivel atmosférico [2], incluso a nivel terrestre [3]. Como resultado, en las nuevas aplicaciones empotradas para sistemas de seguridad y de misión crítica, donde los fallos pueden implicar importantes pérdidas económicas o incluso de vidas humanas, son necesarias la aplicación de técnicas de tolerancia a fallos. Ejemplo de ello, son los saté-

lites de bajo coste desarrollados recientemente [4], sistemas robustos para la industria automotriz [5], entre otros.

Las técnicas más usuales para la mitigación del efecto de los fallos transitorios, conocidas como técnicas de *endurecimiento*, se basan en la redundancia del hardware [6, 7, 8]. En general todas ellas brindan una solución muy efectiva, pero su alto coste puede resultar inconveniente para muchos sistemas empotrados.

Con el fin de reducir el coste de adopción de las técnicas de tolerancia a fallos, en la última década han ido apareciendo soluciones basadas en la redundancia del software, proporcionando además un nivel aceptable de fiabilidad [9, 10]. Se pueden encontrar distintas propuestas para detección de fallos, bien mediante el empleo de redundancia de código a alto nivel [11], o a bajo nivel (instrucciones ensamblador) [12, 13]. En general, las técnicas basadas en sofware resultan más baratas de implementar, aunque conllevan una peor fiabilidad y un impacto negativo en el rendimiento.

En este trabajo se presenta una infraestructura extensible basada en compilador, que permite combinar distintas técnicas hardware y software de tolerancia a fallos (co-diseño hw/sw). Las herramientas desarrolladas facilitan la exploración del espacio de diseño de nuevas aplicaciones de seguridad y de misión crítica, donde la fiabilidad juega un papel tan importante como lo puedan ser el rendimiento o el coste del sistema.

Como caso de estudio para validar la propuesta, se ha diseñado un *front-end* y un *back-end* para el compilador, que da soporte al microprocesador *PicoBlaze* [14]. También se han implementado y evaluado tres técnicas software y se ha estudiado detalladamente el endurecimiento de una aplicación sobre distintas versiones de *PicoBlaze*. Del lado software, se ha empleado una modificación mejorada de la técnica *SWIFT-R* [15], entre otras. Del lado hardware, se han empleado 5 descripciones *RTL* de *Picoblaze* independientes de *FPGA*, variando la selección de estructuras redundantes. Con todo ello, se han llevado a cabo diferentes medidas de los compromisos entre tamaño del código, rendimiento, cobertura a fallos y costes.

El resto del artículo se organiza como sigue. La siguiente sección presenta el modelo de fallos empleado y la terminología empleada. La sección 3 presenta la infraestructura de endurecimiento. La sección 4 describe el caso de estudio comentado, incluyendo los resultados experimentales y finalmente la sección 5 presenta las conclusiones del trabajo.

2. Modelo de fallos y terminología

El presente trabajo se centra en el conocido modelo denominado *SEU — Single Event Upset*. En dicho modelo de fallos acontece una permutación del valor de un determinado bit (*bit-flip*) por cada ejecución de programa a causa de la ionización provocada por la interacción con una partícula cargada. A pesar de su sencillez, este modelo es ampliamente utilizado en la comunidad de tolerancia a fallos dada su cercanía al comportamiento real [11].

Para evaluar la fiabilidad del sistema, clasificamos los fallos inyectados en consonancia con el efecto que producen en la ejecución del programa tal y como propone Reis *et al.* [15]. Si el fallo no impide que el programa continúe su ejecución, pero se producen resultados incorrectos, se cataloga como *Silent Data Corruption* (*SDC*). Si los resultados son los esperados, el fallo se cataloga como *unnecessary for Architecturally Correct Execution* (*unACE*). Finalmente, si produce una finalización anormal del programa, o entra en un bucle infinito, se cataloga como *Hang*. Los efectos *SDC* y *Hang* no son deseables en ningún caso.

Vale la pena mencionar que las técnicas basadas en software introducen redundancia, lo que implica dos hechos importantes a tener en cuenta. Primero, estas técnicas incrementan el tiempo de ejecución de los programas, y por tanto también la probabilidad de ocurrencia de fallo. Segundo, la redundancia incrementa el número de bits presentes en el sistema, incrementando el número de bits susceptibles de fallo. Por tanto, la cobertura frente a fallos ofrecida por una determinada estrategia de endurecimiento está directamente relacionada con el porcentaje de fallos *unACE* y el incremento del tiempo de ejecución.

3. Infraestructura propuesta

3.1. Entorno de endurecimiento software

El entorno de endurecimiento está compuesto por un compilador de endurecimiento *multiobjetivo*, un simulador genérico de instrucciones máquina, en adelante *ISS*[1] y varios *back-ends* y *front-ends*[2] para dar soporte a distintos micros. La figura 1 muestra un esquema general del entorno de endurecimiento.

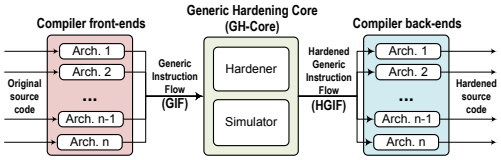

Figura 1: Entorno de endurecimiento de software

Como puede observarse, un cierto *front-end* del compilador toma el código ensamblador de un micro soportado, realiza el análisis léxico, sintáctico y semántico del código y genera un *GIF* (*Flujo Genérico de Instrucciones*). Este flujo representa una abstracción de alto nivel del programa para una arquitectura genérica. Posteriormente, se realizan las tareas de endurecimiento sobre el *GIF* con el módulo *GH-Core* (*Núcleo Genérico de Endurecimiento*), que genera un nuevo *GIF* endurecido o *H-GIF*. Este nuevo flujo se traslada mediante un cierto *back-end* a código nativo de cualquiera de los microprocesadores soportados. Nótese que es posible obtener código endurecido para un micro *A* a partir de las fuentes de un programa para un micro *B*.

Cabe destacar que el módulo de endurecimiento *GH-Core* está basado en una arquitectura de microprocesador genérica, lo que permite diseñar técnicas de tolerancia a fallos con independencia del procesador. La generación del código endurecido es automática y sigue reglas de transformación a nivel de instrucción.

GH-Core tiene dos componente principales: el endurecedor y el simulador (*ISS*). El endurecedor integra las rutinas usuales de mitigación de errores (que pueden ser extendidas). De otra parte, el *ISS* asiste al diseñador en la implementación de dichas rutinas. Esto permite producir automáticamente diferentes análisis en los flujos *GIF* y *HGIF* para comprobar la corrección del proceso de endurecimiento, a la vez que obtener información útil para el proceso de co-diseño: incremento del tiempo de ejecución y del tamaño del código, estimaciones de fiabilidad, etc. Para estimar la fiabilidad obtenida mediante las técnicas software, el *ISS* puede simular *SEUs* realizando permutación de valores de bits aleatorios. Con esto, el conjunto *Hardener+ISS* ofrece un rico conjunto de parámetros de co-diseño que pueden ser usados para realizar una exhaustiva exploración del espacio de diseño.

El endurecedor clasifica de modo especial las instrucciones cuya ejecución implica cruzar la frontera de la *Esfera de Replicación* (*SoR*) [16], que es el dominio lógico de replicación. Utilizando este concepto, es posible incluir/excluir al subsistema de memoria y el banco de registros dentro de la *SoR*, o bien sólo un subconjunto seleccionado de registros. Cuando una instrucción provoca la entrada de un dato en la esfera de replicación (mediante lectura de puerto, carga de valor en registro o lectura de memoria), ésta es etiquetada como *inSoR*; consecuentemente, si se produce una escritura en puerto o en memoria, se etiqueta como *outSoR*. Las fronteras de la *SoR*, y en consecuencia, la cobertura de la protección, puede cambiar según la técnica aplicada.

3.2. *FTUnshades*

El segundo componente de la infraestructura es el entorno *FTUnshades* [17]. Se trata de una plataforma basada en *FPGA* para el estudio de la fiabilidad de sistemas digitales frente a *soft-errors*. El entorno permite configurar la FPGA con una versión real del sistema completo e inyectar fallos, de forma no intrusiva, durante la ejecución de una aplicación. Los *SEUs* se emulan induciendo permutas del valor de ciertos bits escogidos al azar por medio de una reconfiguración dinámica. Una suite de herramientas software permiten comprobar el

[1] Instruction Set Simulator

[2] rutinas de traducción del ensamblador original al genérico y viceversa

diseño, recoger y analizar los resultados de las campañas de inyección de fallos. En este trabajo, *FTUnshades* fue utilizado para validar la cobertura a fallos de las distintas versiones *HW/SW* de los sistemas.

4. Caso de Estudio

Para evaluar la funcionalidad de la infraestructura propuesta, se ha desarrollado un *front-end* y *back-end* para *PicoBlaze*. Así mismo, se ha codificado en VHDL una versión tecnológicamente independiente del microprocesador (*RTL-PicoBlaze*), que es equivalente ciclo a ciclo a la original.

Como parte del caso de estudio, se han implementado tres técnicas software de protección basadas en la Triple Redundancia Modular (*TMR*). Dos de ellas siguen una estrategia local y están destinadas a proteger únicamente las instrucciones aritméticas y lógicas; la tercera es una adaptación de la técnica *SWIFT-R*, que sigue una estrategia global.

El procedimiento de mitigación desarrollado en la primera técnica *TMR1* consta de los siguientes pasos: 1 – Identificación de los nodos (*bloques básicos*), sub-nodos y construcción del *Grafo de Control de Flujo CFG* del programa. 2 – Triplicación de instrucciones deseadas. Cada instrucción redundada opera con copias de registros. 3 – Inserción de votadores por mayoría y procedimientos de recuperación para los registros protegidos. 4 – Liberación de copias redundantes de registros después de que cada votador haya verificado su corrección. 5 – Opcionalmente, durante el proceso de endurecimiento, se pueden insertar votadores y procedimientos de recuperación si no hay suficientes registros para realizar réplicas y hace falta liberar estos recursos.

La segunda técnica implementada *TMR2* consiste en detectar y corregir fallos en la memoria de datos mediante el cómputo duplicado de las instrucciones seleccionadas más un tercer cómputo opcional en caso de discrepancia. La Fig. 2 muestra un ejemplo de un programa *PicoBlaze* endurecido con *TMR1* y *TMR2*.

SWIFT-R es un método global enfocado en la recuperación frente a fallos en la sección de

Figura 2: Endurecimiento con *TMR1* y *TMR2*

datos que está muy ligado al banco de registros del micro. Nuestra implementación del método puede resumirse como:

1 – Identificación de *nodos*, *sub-nodos* y construción del *CFG*. 2 – Triplicación de datos la primera vez que entran en la esfera de redundancia *SoR* (después de instrucciones *inSoR*). En este caso, sólo el banco de registros se considera incluido en la *SoR*; no así la de memoria, porque asumimos que tiene su propio mecanismo de protección [15]. Por lo tanto, para cada instrucción *inSoR*, se crean 2 copias adicionales del valor que ingresa a la esfera, sin acceder de nuevo a memoria o puertos. 3 – Triplicación de las instrucciones que operen con datos. 4 – Comprobación de la consistencia (insertando votadores y prodecimientos de recuperación) de los datos que intervienen en las instrucciones siguientes: tipo *outSoR* y las localizadas antes de un salto condicional (afectan a los *flags* y pueden ocasionar un salto incorrecto). 5 – Los registros redundantes pueden liberarse si no se usan más (implica un análisis detallado del *CFG*).

Nótese que *TMR1* guarda las copias de los registros sólo hasta su comprobación, mientras que *SWIFT-R* las mantiene mucho más, hasta que no se vuelvan a usar más.

Se han realizado 2 experimentos para el caso de estudio. El primero evalúa las técnicas de redundancia implementadas con un *benchmark* de aplicaciones: ordenación-burbuja (*bub*), división, multiplicación, potenciación escalar (*div*, *mult*, *pow*), *Fibonacci* (*fib*), Máximo Común Divisor (*gcd*), suma y multiplicación de matrices (*madd*, *mmult*). El segundo experimento muestra las posibilidades del entorno en el co-diseño de sistemas robustos.

4.1. Evaluación de las técnicas software

Usando la infraestructura, se ha endurecido de forma automática cada programa del *benchmark* aplicando las tres técnicas software.

La figuras 3 y 4 muestran los incrementos obtenidos (normalizados con respecto a la versión sin endurecer) en tamaño de código y tiempo de ejecución.

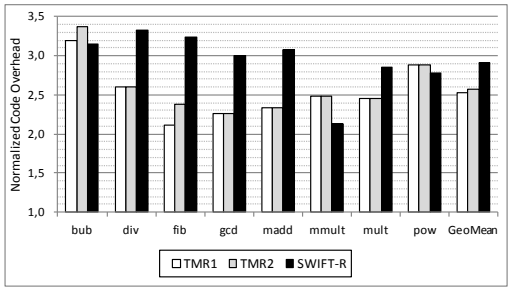

Figura 3: Incremento normalizado del código

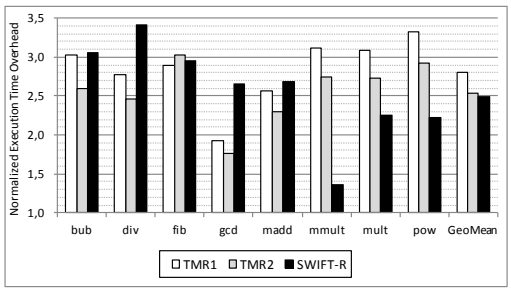

Figura 4: Incremento normalizado del tiempo de ejecución

TMR1 y *TMR2* presentan casi el mismo impacto sobre el código, la media geométrica de todos los programas del incremento normalizado de codigo y tiempo de ejecución es de $\times$2,52 y $\times$2,58 respectivamente, mientras que la técnica *SWIFT-R* es ligeramente mayor en código $\times$2,92. Sin embargo, con respecto a la figura 4, *SWIFT-R* ofrece el menor impacto en el tiempo de ejecución ($\times$2,50), mientras que *TMR1* y *TMR2* lo degradan significativamente ($\times$2,81 y $\times$2,54). Estos análisis de incrementos pueden motivar importantes decisiones de diseño en función de las restricciones de cada aplicación.

El siguiente experimento evalúa la fiabilidad ofrecida por las técnicas. Para ello, sobre cada aplicación (original y endurecida) se han llevado a cabo 5000 ejecuciones en el simulador (*ISS*). En consonancia con el modelo de fallos, sólo se simula un *SEU* en el banco de registros por cada ejecución. La figura 5 muestra los porcentajes de fallos para cada programa.

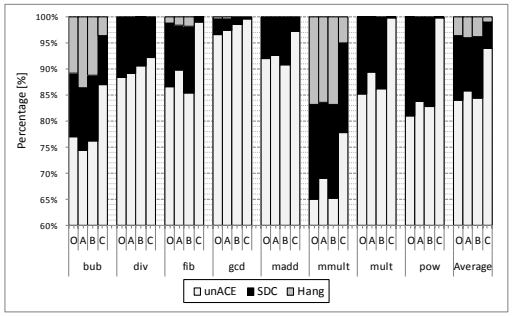

Figura 5: Clasificación de fallos en porcentaje para la versión sin endurecer (O), *TMR1* (A), *TMR2* (B), *SWIFT-R* (C) usando el simulador (campaña de test ejecutada sobre el banco de registros)

En suma los porcentajes *unACE* son: 83,95 % para la versión sin endurecer, 85,68 % para *TMR1*, 84,44 % para *TMR2* y 94,01 % para *SWIFT-R*. Estos porcentajes representan una estimación de la cobertura a fallos, ya que el *ISS*, al ser genérico no tiene en cuenta la micro–arquitectura del procesador. No obstante, se trata de una información valiosa que permite comparar la calidad de cada una de las técnicas, sin necesidad de recurrir a medidas más precisas (nótese que *SWIFT-R* ofrece una mejor cobertura que *TMR1* y *TMR2*). Para validar las estimaciones del *ISS*, se ha utilizado *FTUnshades* con una campaña de inyección de fallos similar a la anterior. La Fig. 6 muestra los resultados obtenidos. En este caso, el porcentaje de fallos *unACE* ha sido de 89,50 % para la versión sin endurecer, 90,56 % para *TMR1*, 89,77 % para *TMR2* y 96,39 % para *SWIFT-R*. Estos resultados confirman las pruebas anteriores de simulación. Aunque los porcentajes obtenidos por emula-

ción son ligeramente superiores a los de simulación, mantienen la misma tendencia.

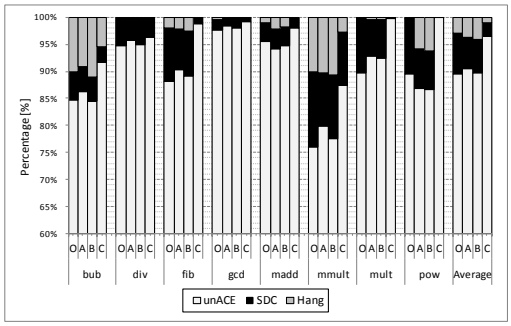

Figura 6: Clasificación de fallos en % para la versión sin endurecer (O), *TMR1* (A), *TMR2* (B), *SWIFT-R* (C) usando *FTUnshades* (Campaña de test ejecutada sobre el banco de registros)

Los resultados de cobertura junto con los incrementos reflejados deberán ser tenidos en cuenta en las decisiones sobre el diseño, ya que representan un compromiso entre tamaño del código, rendimiento y fiabilidad. Si el tiempo de ejecución es crítico, para maximizar la cobertura a fallos se descartarían las técnicas más lentas, seleccionando las más apropiadas para las restricciones de diseño.

4.2. Espacio de *Co–Diseño*

Con la infraestructura propuesta es posible seguir diferentes estrategias de co-diseño. Para demostrar las posibilidades se ha realizado un análisis en profundidad del algoritmo de multiplicación de matrices en *Picoblaze*, implementando distintos niveles de endurecimiento tanto en hardware como en software. En esta prueba, *SWIFT-R* se ha aplicado selectivamente sólo a diferentes combinaciones de registros en el software (*mmult* sólo usa los registros *0*, *A*, *D*, *E* y *F*). Fig. 7 muestra los incrementos normalizados de tiempo y tamaño para distintas versiones endurecidas de *mmult*.

Para ayudar al diseñador a priorizar qué registros hay que endurecer y cuáles no, el *ISS* ofrece información sobre el *tiempo de vida* de cada uno y el número de accesos que recibe. El cuadro 1 refleja esta información. El *tiempo*

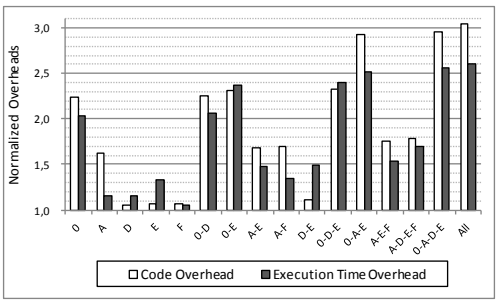

Figura 7: Incrementos de tamaño y tiempo de ejecución para *mmult* usando hardening selectivo

Cuadro 1: Uso de registros en *mmult*

Registro	#Escrituras	#Lecturas	#Lec/Esc	Tiempo vida [%]
0	340	357	183	55,3%
A	20	83	45	85,7%
D	27	135	108	48,7%
E	27	135	108	52,2%
F	9	9	27	83,3%

de vida expresa el tiempo total durante el cual un registro retiene un dato válido, que está directamente relacionado con su sensibilidad a fallos transitorios. El número de veces que se accede a un registro, está ligado al impacto que tendrá endurecimiento en el tamaño del código y el rendimiento del programa. Ambos factores deben tenerse en cuenta para diseñar la estrategia de endurecimiento.

La protección del sistema se ha complementado aplicando *TMR* por hardware a diferentes conjuntos de registros. Se han generado las siguientes versiones de *Picoblaze*: **P0:** sin endurecer; **P1:** contador de programa, flags, y el puntero de pila; **P2:** registros del *pipeline*; **P3:** *P1* + *P2*; **P4:** versión totalmente endurecida (*P3* + banco de registros).

Para evaluar la fiabilidad global del sistema se han preparado las siguientes pruebas: para cada sistema (endurecido selectivamente en software y hardware), se ha ejecutado una campaña de inyección en *FTUnshades*. Cada campaña hace ataques selectivos sobre subconjuntos de registros. Cada subconjunto sufre 5000 *SEUs* (1 por ejecución) en ciclos de reloj escogidos al azar. La figura 8 presenta la clasificación porcentual de fallos en cada sistema. Estos resultados son la suma ponderada de los

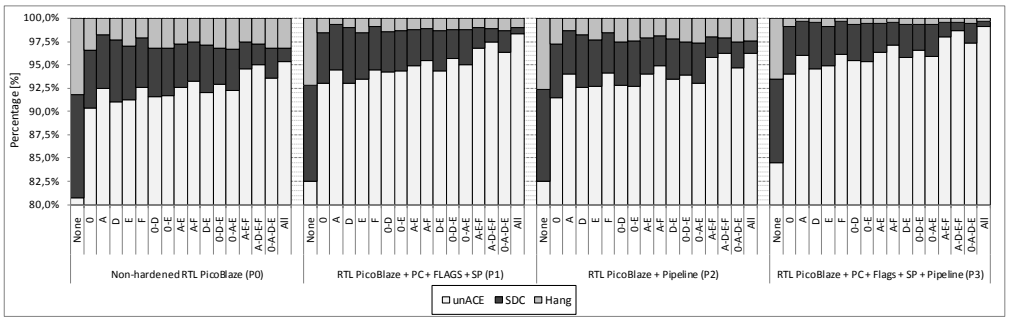

Figura 8: Clasificación porcentual de fallos para cada test versión del procesador ($P0 \ldots P3$)

obtenidos en los ataques selectivos, suponiendo idéntica probabilidad de fallos en cada bit.

El análisis de estos datos permite encontrar soluciones de compromiso entre los distintos parámetros del diseño que mejor se acomoden con las especificaciones del sistema.

Nótese que *SWIFT-R* ofrece un considerable aumento de la fiabilidad, incluso en la versión de *Picoblaze* no endurecida (hasta 95,38 % de fallos *unACE* en caso de protección de todos los registros). Estos resultados son sensiblemente mejores que los obtenidos con cada versión de *Picoblaze* endurecida únicamente por hardware (versiones *none* en la figura). La combinación de la técnica SWIFT-R con la versión P1 aumenta la fiabilidad hasta alcanzar 98,32 % de fallos *unACE*. Como demuestran los resultados, el endurecimiento de los registros del pipeline no mejora la cobertura, aún siendo más numerosos que los protegidos en la versión *P1*. Los resultados mejoran al aumentar el número de registros protegidos (*P3*), llegando a una completa cobertura en la versión *P4* (estos resultados no se muestran al obtenerse siempre un 100 % de *unACEs*).

Para obtener una mejor estimación de los costes en área, cada versión se ha sintetizado con *Xilinx XST 10.1*. La figura 9, muestra estos costes normalizados con respecto a la versión *P0* en términos de *Flip/Flops* y *Latches*, primitivas(*MUXs*, *LUTs*) y RAM (tanto distribuida como en bloques); también ilustra el porcentaje de fallos *unACE* al aplicar *SWIFT-R* a todos los registros del banco.

Cabe destacar que los costes se incrementan considerablemente cuando se endurecen los registros del *pipeline* (*P2* y *P3*), mientras que la fiabilidad mejora tímidamente o incluso empeora si comparamos con aproximaciones más económicas (*P2+SWIFT-R*). Para *P4*, los altos costes hardware podrían ocasionar que el diseño fuera desestimado, aún siendo la fiabilidad del 100 %.

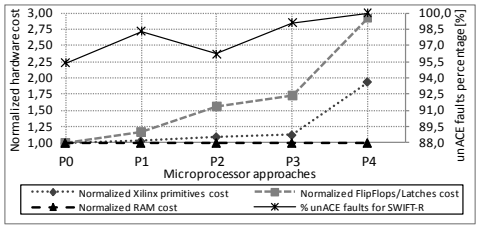

Figura 9: Costes hardware normalizados y porcentaje de fallos *unACE* para cada versión del micro

5. Conclusiones

Este trabajo presenta una infraestructura de endurecimiento dirigida por compilador para el codiseño de nuevas aplicaciones de los sistemas empotrados, relacionadas con la seguridad y los sistemas de misión crítica. Con ella se facilita la exploración del espacio de diseño entre técnicas de tolerancia a fallos puramentes hardware y las puramente software mediante la evaluación del compromiso entre ciertas restricciones de diseño, fiabilidad y coste hardwa-

re. Las ventajas de la aproximación se ilustran por medio de un caso de estudio exhaustivo.

Agradecimientos

Este trabajo ha sido financiado por los siguientes proyectos: *'RENASER'* (ESP2007-65914-C03-03) del Ministerio de Ciencia y Educación; y *'Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware'* (GV/2009/098) (Generalitat Valenciana).

Referencias

[1] R Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Tran. on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[2] R Edwards, C Dyer, and E Normand. Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics. In *IEEE Radiation Effects Data Workshop*, pages 1–5, 2004.

[3] R Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121, April 2002.

[4] Del Corso et al. Architecture of a small low-cost satellite. In *DSD 2007: 10TH EUROMICRO Conf. Digital System Design Architectures, Methods and Tools*, pages 428–431. Drager, 2007.

[5] D. Skarin, M. Sanfridson, and J. Karlsson. Impact of soft errors in a brake-by-wire system. In *IEEE Workshop on Silicon Errors in Logic*, April 2007.

[6] TM Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *32nd Annual Int. Symp. on Microarchitecture*, pages 196–207, 1999. Haifa, Israel, Nov 16-18, 1999.

[7] Naseer Riaz et al. Analysis of soft error mitigation techniques for register files in IBM Cu-08 90nm technology. In *49th IEEE Int. Midwest Symposium on Circuits and Systems*, pages 515–519, 2006.

[8] A Mahmood and EJ McCluskey. Concurrent error-detection using watchdog processors. *IEEE Trans. Comput.*, 37(2):160–174, Feb 1988.

[9] N. Oh, S. Mitra, and E. J McCluskey. (EDI)-I-4: error detection by diverse data and duplicated instructions. *IEEE Tran. on Computers*, 51(2):180–199, 2002.

[10] M. Rebaudengo et al. A new software-based technique for low-cost Fault-Tolerant application. *Annual Reliability and Maintainability Symposium, 2003 Proceedings*, pages 25–28, 2003.

[11] M. Rebaudengo et al. A source-to-source compiler for generating dependable software. *First IEEE International Workshop on Source Code Analysis and Manipulation, Proceedings*, pages 33–42, 2001.

[12] N. Oh, P.P Shirvani, and E. J McCluskey. Control-flow checking by software signatures. *IEEE Tran. on Rel.*, 51(1), 2002.

[13] G. A Reis et al. SWIFT: software implemented fault tolerance. *CGO: Int. Sym. Code Gen. and Opt.*, pages 243–254, 2005.

[14] K. Chapman. *PicoBlaze KCPSM3. 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II Pro.* Xilinx, 2003.

[15] G. A Reis, J. Chang, and D. I August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, 2007.

[16] SK Reinhardt and SS Mukherjee. Transient fault detection via simultaneous multithreading. In *27th Int. Symp. on Comp. Arch.*, pages 25–36, 2000.

[17] H. Guzman-Miranda, M.A. Aguirre, and J. Tombs. Noninvasive fault classification, robustness and recovery time measurement in microprocessor-type architectures subjected to radiation-induced errors. *IEEE Tran. on Instrumentation and Measurement*, 58(5), May 2009.