Prototipado Rápido de Sistemas Empotrados Tolerantes a Radiación en FPGA

Felipe Restrepo-Calle[1], Antonio Martínez-Álvarez[1], F.R. Palomo[2]
H. Guzmán-Miranda[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

{frestrepo, amartinez, sergio}@dtic.ua.es, {hipolito, rogelio, aguirre}@gte.us.es

Resumen

La creciente capacidad de integración de las *FPGA* está convirtiendo estos dispositivos en la plataforma preferida para el prototipado rápido de sistemas digitales complejos. Por otro lado, a medida que la tecnología se reduce, cobra importancia la protección de los sistemas frente a los *fallos transitorios* inducidos por radiación (por ejemplo los *Single Event Upsets*). En este trabajo se presenta una nueva aproximación de prototipado rápido para el codiseño de sistemas empotrados robustos usando *FPGA*. Dicha aproximación está soportada por una plataforma de endurecimiento que permite combinar técnicas de tolerancia a fallos basadas en software con técnicas basadas en hardware, obteniendo diferentes configuraciones hardware/software con diferentes niveles de compromiso entre restricciones de diseño, fiabilidad y coste. Como caso de estudio, se han desarrollado varios sistemas empotrados tolerantes a radiación basados en una versión del microprocesador *PicoBlaze* independiente de tecnología.

1. Introducción

En las últimas décadas la miniaturización progresiva de los componentes electrónicos ha ocasionado avances importantes en las prestaciones de los microprocesadores. En contrapartida, a medida que la tecnología permite disminuir el tamaño de los componentes básicos, se reducen de forma considerable los márgenes de alimentación y ruido, lo que provoca que dichos dispositivos sean cada vez menos fiables y más susceptibles a *fallos transitorios* inducidos por radiación. Dichos fallos no suelen causar un fallo permanente en el sistema, pero sí la ejecución incorrecta de un programa al verse afectadas las transferencias de señales o los valores almacenados [1, 2].

Tradicionalmente se han utilizado técnicas de hardware redundante para proteger o "endurecer" (*hardening*) los sistemas frente a *fallos transitorios*. Estas técnicas hacen uso tanto de estructuras de bajo nivel (códigos de corrección de errores *ECC — Error-Correcting Code*, triple redundancia modular *TMR — Triple Modular Redundancy*), como de unidades funcionales [3], coprocesadores [4], incluso bloques hardware complejos como los disponibles en las modernas arquitecturas multihebra y multi-núcleo [5, 6, 7]. No obstante, aunque las técnicas hardware proveen una solución muy efectiva, tienen el inconveniente de que su alto coste de implementación las hace inviables en muchos ocasiones.

Motivada por la necesidad de soluciones de bajo coste que lidien con este problema, la comunidad científica ha propuesto distintas alternativas basadas en el endurecimiento de software [8, 9, 10, 11, 12]. Estas nuevas soluciones son más económicas que sus duales hardware, pero conllevan un descenso en la fiabilidad y el rendimiento (la redundancia software implica ejecutar instrucciones adicionales).

Para amplios segmentos del mercado de los sistemas empotrados, y fruto de la creciente

sensibilidad de los microprocesadores, la fiabilidad se está convirtiendo en un factor de diseño tan importante como lo pueden ser el coste, el consumo o el rendimiento. Para este tipo de aplicaciones, las soluciones óptimas se obtienen combinando aspectos hardware y software de distintas técnicas de protección. Algunas de las más recientes aproximaciones híbridas han demostrados resultados prometedores [13, 14]. Sin embargo, hasta el momento son demasiado genéricas y carecen de la necesaria flexibilidad para obtener buenos compromisos entre la cobertura frente a fallos y el resto de los parámetros de diseño.

En este contexto, es importante contar con las herramientas adecuadas que permitan a los diseñadores explorar fácilmente el espacio de diseño para encontrar la mejor configuración hardware/software que satisfaga los requisitos del sistema. Además, es necesario contar con entornos de prototipado que permitan verificar y validar los sistemas cumpliendo con las severas restricciones de diseño y *"time-to-market"* de las aplicaciones actuales.

En este trabajo, las *FPGAs* son utilizadas como plataforma de desarrollo y verificación de sistemas robustos hardware/software. Las técnicas de mitigación de fallos se aplican a alto nivel de abstracción, lo que permite validarlas para distintas tecnologías de *FPGAs* y *ASICs*.

Nuestra propuesta está soportada por dos *suites* de herramientas: un *entorno de endurecimiento de software* [15] para implementar, evaluar y aplicar automáticamente técnicas software de tolerancia frente a fallos; y una herramienta de emulación de fallos basada en *FPGA* llamada *FTUnshades* [16], que permite evaluar distintas métricas de fiabilidad sobre una implementación física del sistema completo. Como caso de estudio para validar la propuesta, se han *co-endurecido* varias aplicaciones empotradas basadas en el conocido soft–core *PicoBlaze* [17]. Del lado del software, la técnica *SWIFT-R* [18] ha sido implementada y aplicada automáticamente a varios programas de prueba; mientras que del lado del hardware, se ha utilizado una descripción independiente de tecnología de *PicoBlaze* (*RTL-PicoBlaze*) para generar cinco versiones del procesador con distintos niveles de redundancia hardware.

2. Modelo de Fallos y Terminología

El presente trabajo se centra en el conocido modelo denominado *SEU*[1]. En dicho modelo de fallos, acontece una sola permutación del valor de un determinado bit (*bit–flip*) durante la ejecución del programa. A pesar de su sencillez, este modelo es ampliamente utilizado en la comunidad de tolerancia a fallos dada su cercanía al comportamiento real [12].

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. [18]. 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 el programa 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 hacer notar 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 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.

[1] *Single Event Upset*

3. Plataforma para el Desarrollo de Sistemas Empotrados Robustos

3.1. Entorno de endurecimiento software

El entorno propuesto constituye una herramienta para el desarrollo de software endurecido. Permite diseñar e implementar técnicas de tolerancia a fallos basadas en software, además de su aplicación a los programas de forma automática. El entorno está compuesto por un compilador de endurecimiento *multiobjetivo*, un simulador genérico de instrucciones máquina, en adelante *ISS* (*Instruction Set Simulator*) y varios *front-ends* y *back-ends* (rutinas de traducción del ensamblador original al genérico y viceversa) para dar soporte a distintos procesadores. La figura 1 presenta un esquema general del entorno.

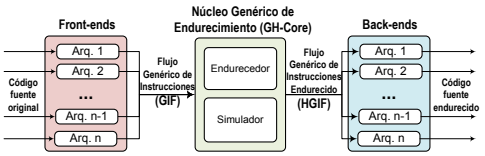

Figura 1: Entorno de endurecimiento de software

Un cierto *front-end* del compilador toma el código ensamblador de alguno de los microprocesadores 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*. Por último, este nuevo flujo se traslada mediante un *back-end* a código nativo de cualquiera de los microprocesadores soportados. Estos *back-ends* realizan transformaciones directas entre el conjunto de instrucciones de la arquitectura genérica y el repertorio de instrucciones del microprocesador seleccionado. Nótese que es posible obtener código endurecido para un procesador A a partir de las fuentes de un programa para un procesador B.

La arquitectura de microprocesador genérica reúne los elementos comunes de diferentes arquitecturas con el fin de facilitar el diseño e implementación de las técnicas de tolerancia a fallos actuales basadas en software con independencia del microprocesador. La generación del código endurecido es automática y sigue reglas de transformación a nivel de instrucción. Para esto, la arquitectura genérica posibilita la identificación del *Grafo de Control de Flujo* (*CFG*) de los programas, al igual que permite realizar cambios en tiempo de compilación.

El *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 cobertura frente a fallos, etc. Para estimar la fiabilidad obtenida mediante las técnicas software, el *ISS* puede también simular *SEUs* por medio de un *bit-flip* en alguno de los bits del banco de registros del microprocesador. Con esto, el conjunto *Hardener+ISS* ofrece un rico conjunto de parámetros de codiseño que pueden ser usados para realizar una exhaustiva exploración del espacio de diseño del lado del software.

El endurecedor clasifica de modo especial las instrucciones cuya ejecución implica cruzar la frontera de la *Esfera de Replicación* (*SoR*) [19]. Esta esfera define el dominio de protección de la técnica en cuestión. 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 una lectura de un puerto, carga de valor en registro o lectura de memoria), dicha instrucción es etiquetada como *inSoR*; consecuentemente, si se produce una escritura en un puerto, o acceso a memo-

ria para escritura, se etiqueta como *outSoR*. Las fronteras de la esfera de replicación *SoR*, y en consecuencia la cobertura de la protección, puede cambiar según la técnica aplicada.

3.2. Herramienta de emulación de *SEUs*: *FTUnshades*

El segundo componente de la infraestructura es el entorno *FTUnshades* [20, 16]. Se trata de una plataforma basada en *FPGA* para el estudio de la fiabilidad de sistemas digitales frente a *fallos transitorios*. 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 reconfiguración parcial dinámica. Una suite de herramientas software permite comprobar el diseño, recoger y analizar los resultados de las campañas de inyección de fallos. En este trabajo, *FTUnshades* fue utilizado para validar la cobertura frente a fallos de las distintas versiones *HW/SW* de los sistemas.

4. Caso de Estudio

Con el fin de validar nuestra propuesta, hemos diseñado y evaluado varios prototipos de sistemas empotrados tolerantes a radiación basados en el microprocesador *PicoBlaze*.

PicoBlaze es un soft-micro de 8 bits ampliamente utilizado en sistemas empotrados basados en *FPGA*. En este trabajo, se ha desarrollado una versión independiente de la tecnología de *PicoBlaze* (*RTL PicoBlaze*), la cual permite validar la propuesta tanto para tecnologías *ASIC* como *FPGAs* de distintos fabricantes y tecnologías. Esta versión es *RTL* equivalente a la versión original del microprocesador (*PicoBlaze-3*). Sus características principales son: 16 registros de propósito general (de 8 bits cada uno), 1K de memoria de programa, *ALU* de 8 bits con indicadores de *ZERO* y *CARRY*, y memoria *Scratchpad RAM* de 64 posiciones.

Todos los programas de prueba utilizados han sido endurecidos aplicándoles la técnica *SWIFT-R*. Ésta es una técnica general que busca recuperar fallos de la sección de datos, principalmente relacionados con el banco de registros del microprocesador. Nuestra implementación de dicha técnica se resume a continuación.

1. Construcción del grafo de control de flujo del programa (*CFG*).
2. Triplicación de los datos la primera vez que entran a la *Esfera de Replicación* – *SoR* (esto es, después de instrucciones clasificadas como *inSoR*). En este caso únicamente el banco de registros se considera dentro de la *SoR*, mientras que el subsistema de memoria no, ya que se asume que éste ya tiene sus propios mecanismos de protección [18]. Por lo tanto, para cada instrucción clasificada como *inSoR* (lectura de un puerto de entrada, lectura desde la memoria, carga de un valor en un registro), se crean dos copias adicionales, copiando el valor del registro sin repetir accesos a memoria o a los puertos.
3. Triplicación de las instrucciones que realizan alguna operación sobre los datos (e.g. instrucciones aritméticas, lógicas, desplazamiento/rotación).
4. Verificación de la consistencia de los datos involucrados en las siguientes instrucciones (la verificación se hace insertando votadores por mayoría antes de ejecutar las instrucciones a saber):
   - Instrucciones clasificadas como *outSoR*. Por ejemplo, almacenar un valor en una posición de memoria, o escribir un valor en un puerto.
   - Aquellas instrucciones localizadas justo antes de un salto condicional. Esta verificación es necesaria ya que dichas instrucciones pueden afectar los indicadores de la *ALU* (*Zero, Carry*), por lo que si algún valor está alterado, el resultado de los indicadores puede ser erróneo también, lo cual provocaría la ejecución de un salto de forma incorrecta.

5. Los registros-copia únicamente pueden ser liberados en los siguientes casos:
    - Si el registro no va a ser usado más en lo que resta del programa.
    - Si la próxima vez que el registro sea usado, será sobrescrito. Nótese que esta condición implica un análisis detallado del *CFG* para prevenir la pérdida de consistencia de los datos.

5. Experimentos y Resultados

Como banco de pruebas para los experimentos se han utilizado los siguientes programas: ordenación burbuja (*bub*), división escalar (*div*), Fibonacci (*fib*), máximo común divisor (*gcd*), suma de matrices (*madd*), multiplicación de matrices (*mmult*), multiplicación escalar (*mult*) y potenciación (*pow*).

La Fig. 2 presenta el incremento del tamaño del código y del tiempo de ejecución, obtenido por el *ISS* una vez aplicada la técnica *SWIFT-R* en cada uno de los programas de prueba. Estos resultados están normalizados con respecto a las versiones sin endurecer. En este caso, la media geométrica – *GeoMean* (calculada considerando todos los programas de prueba) del incremento normalizado del código es $\times$3,05 y del incremento normalizado del tiempo de ejecución es $\times$2,70.

Del lado hardware, la estrategia de tolerancia a fallos fue complementada mediante el endurecimiento incremental de los recursos del microprocesador. Para ello se aplicó manualmente la técnica conocida como triple redundancia modular (*TMR*), a distintos subconjuntos del microprocesador, obteniéndose las siguientes versiones:

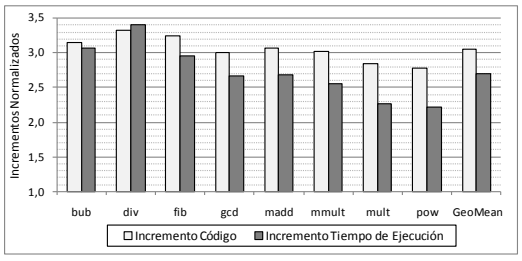

Figura 2: Incremento normalizado del tamaño del código y el tiempo de ejecución para *SWIFT-R*

- *P0: RTL Picoblaze* sin endurecer.
- *P1:* microprocesador con redundancia hardware para el contador de programa (*PC*), indicadores de la *ALU* (Flags), y el puntero de pila (*SP*).
- *P2:* redundancia hardware para todos los registros del *Pipeline*.
- *P3:* redundancia hardware para el *PC*, *Flags*, *SP*, y *Pipeline*.
- *P4:* protegido completamente, es decir, con redundancia hardware para el banco de registros, *PC*, *Flags*, *SP*, y *Pipeline*.

Con el fin de evaluar la fiabilidad de las diferentes configuraciones *HW/SW* del sistema se ha utilizado la herramienta *FTUnshades*. Primero se llevó a cabo un experimento para calibrar la herramienta, calculando el número mínimo de *SEUs* que era necesario inyectar durante una campaña de inyección de fallos, para garantizar que los resultados obtenidos fueran lo suficientemente precisos. En este sentido, se realizaron numerosas campañas de prueba variando el número de *SEUs* inyectados en el banco de registros de forma incremental. Este experimento se realizó con el microprocessor *P0* ejecutando los programas sin endurecer (el peor de los casos). Los resultados muestran que el intervalo de confianza del 95 % es menor que $\pm$1,0 % por encima de 5000 *SEUs* inyectados.

En un segundo experimento, se ha ejecutado una campaña de inyección de fallos para cada uno de los programas de prueba, versión sin endurecer (*O*) y versión endurecida (*H*), y para cada versión del procesador (*P0* ... *P4*). Cada una de las campañas de inyección de fallos realiza ataques selectivos a diferentes conjuntos de registros del microprocesador: banco de registros (128 bits - 65,0 % de los registros del procesador), *PC* (10 bits - 5,1 %), *Flags* (2 bits - 1,0 %), *SP* (5 bits - 2,5 %), *Pipeline* (52 bits - 26,4 %). Además, para cada uno

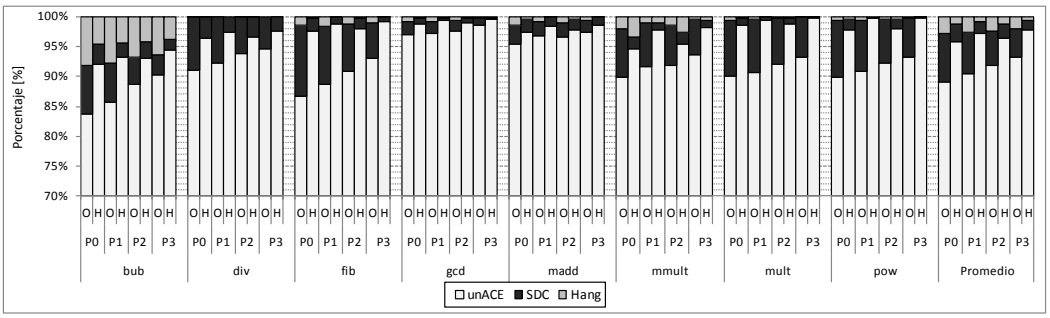

Figura 3: Clasificación de fallos en porcentaje para las versiones sin endurecer (O) y SWIFT-R (H) de todos los programas de prueba, ejecutándose en cada una de las versiones del microprocesador ($P0 \ldots P3$)

de estos conjuntos de registros, se han emulado 5000 *SEUs* (uno por ejecución) en un ciclo de reloj seleccionado aleatoriamente durante el tiempo de ejecución del programa. Finalmente, los resultados han sido clasificados como *unACE*, *SDC*, y *Hang*. La Fig 3 presenta la clasificación de los fallos obtenidos para cada prototipo. Nótese que estos resultados han sido obtenidos calculando el promedio ponderado de los resultados de los ataques selectivos a los conjuntos de registros mencionados anteriormente, asumiendo la misma probabilidad de fallo para cada uno de los bits del microprocesador. Los resultados para la versión del procesador *P4* no se presentan en la Fig. 3 porque 100 % de los fallos inyectados fueron clasificados como *unACE*, como se esperaba.

Como puede observarse, la técnica *SWIFT-R* ofrece un incremento de fiabilidad considerable, incluso para la versión sin endurecer del hardware *P0* (en promedio, hasta el 95,88 % de fallos *unACE*). En promedio, el endurecimiento con *SWIFT-R* ofrece mejor resultado que el endurecimiento sólo por hardware (sin proteger el software), excepto para la versión *P4*. Como demuestran los resultados, el endurecimiento de los registros del *pipeline* no mejora la cobertura frente a fallos, aún siendo más numerosos que los protegidos en la versión *P1*. Por esto, entre las configuraciones híbridas, cabe destacar que combinando *SWIFT-R* con protección hardware aplicada sólo a algunos registros, tales como el *PC*, *Flags*, y *SP* (versión *P1* del procesador), la fiabilidad se incrementa notoriamente (en promedio, hasta 97,18 % de fallos *unACE*).

Para obtener una estimación de los costes en área, cada versión del procesador fue sintetizada usando *Xilinx XST v10.1*, y los resultados expresados en términos de: *Flip/Flops* y *Latches*, primitivas *Xilinx* (*MUX*, *LUT*, etc.), y *RAM* (distribuidas y en bloques). La Fig 4, por un lado, presenta los costes hardware de cada versión del microprocesador normalizados con respecto a la versión no endurecida - *RTL Picoblaze* (*P0*); por otra parte, también ilustra, en un eje secundario, el porcentaje de fallos *unACE* para las versiones sin endurecer y *SWIFT-R* de los programas de prueba (en promedio). Esta figura permite observar cómo se afectan la fiabilidad y los costes conjuntamente en cada una de las versiones del microprocesador.

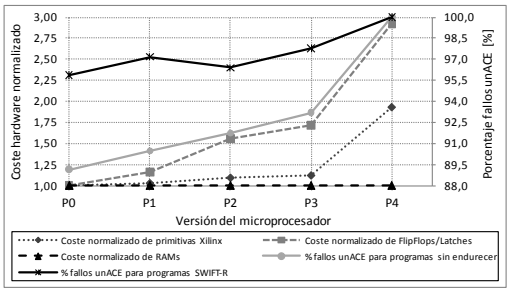

Figura 4: Costes hardware normalizados y porcentaje de fallos *unACE* para cada versión del procesador

Cabe destacar que los costes hardware se incrementan considerablemente al endurecer los registros en el *pipeline* (versiones *P2* y *P3*). Sin embargo, la fiabilidad solo mejora ligeramente en estos casos, o incluso empeora si se compara con aproximaciones más económicas (*P2+SWIFT-R*). En el caso de *P4*, los altos costes hardware podrían resultar inconvenientes para muchas aplicaciones, aún alcanzando una fiabilidad del 100 %.

Por último, el conjunto de resultados obtenidos (incrementos de tamaño de código y tiempo de ejecución, fiabilidad, costes) habilita al diseñador para decidir cual de las configuraciones *HW/SW* es la más apropiada de acuerdo con los requisitos de la aplicación que esté diseñando. Por ejemplo, en este caso podrían resultar apropiados aquellos prototipos con el microprocesador con redundancia hardware en el *PC*, *Flags*, y *SP* (*P1*), combinado con los programas *SWIFT-R*, ya que estos prototipos ofrecen un alto nivel de fiabilidad (97,18 % de fallos *unACE*) con costes aceptables (costes hardware bajos, incremento del tiempo de ejecución ×2,70, e incremento del tamaño del código ×3,05). No obstante, en otros casos podría ser preferible aplicar una técnica más ligera del lado del software e incrementar la protección y los costes del lado hardware.

6. Conclusiones y Trabajo Futuro

Este trabajo presenta una aproximación de prototipado rápido para el diseño de sistemas empotrados tolerantes a radiación usando *FPGA*. Esta aproximación está soportada por una plataforma flexible de endurecimiento, la cual facilita la evaluación del compromiso entre especificaciones de diseño, requisitos de fiabilidad, coste, etc. Esta estrategia de prototipado rápido permite a los diseñadores explorar con facilidad el espacio de diseño entre las técnicas de tolerancia a fallos basadas en software y aquellas basadas en hardware. Las ventajas de las implementaciones híbridas *HW/SW* son ilustradas por medio de un caso de estudio. En este contexto, se han desarrollado varios prototipos de sistemas empotrados robustos (basados en una implementación *RTL* del soft-micro *PicoBlaze*). Como resultado, esta nueva estrategia sugiere la implementación de tareas automáticas de *co–endurecimiento* dentro de la plataforma propuesta y abre un panorama interesante en la exploración del espacio de diseño. Asimismo, como trabajo futuro, el entorno de endurecimiento de software será extendido para soft-cores de 32 bits, como *MicroBlaze* y *LEON3*.

Agradecimientos

Este trabajo ha sido financiado por los siguientes proyectos: *'RENASER'* (`ESP2007-65914-C03-03`) del Plan Nacional de Investigación 2007 del Ministerio de Ciencia y Educación; y el proyecto de investigación *'Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware'* (`GV/2009/098`) (Generalitat Valenciana, España).

Referencias

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

[2] P Shivakumar, M Kistler, SW Keckler, D Burger, and L Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks, Proceedings*, pages 389–398, 2002.

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

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

[5] TN Vijaykumar, I Pomeranz, and K Cheng. Transient-fault recovery using

simultaneous multithreading. In *29th Annual International Symposium on Computer Architecture*, pages 87–98, 2002. Anchorage, AK, May 25-29, 2002.

[6] SS Mukherjee, M Kontz, and SK Reinhardt. Detailed design and evaluation of Redundant Multithreading alternatives. In *29th Int. Symp. on Computer Architecture*, pages 99–110, 2002. AK, May 25-29, 2002.

[7] MA Gomaa, C Scarbrough, TN Vjaykumar, and I Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE MICRO*, 23(6):76–83, Nov-Dec 2003.

[8] B. Nicolescu, Y. Savaria, and R. Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *Ieee Transactions on Nuclear Science*, 51(6):3510–3518, 2004.

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

[10] N. Oh, P. P Shirvani, and E. J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1), 2002.

[11] G. A Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I August. SWIFT: software implemented fault tolerance. *CGO 2005: Int Symp. on Code Gen. and Optimization*, pages 243–254, 2005.

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

[13] GA Reis, J Chang, N Vachharajani, SS Mukherjee, R Rangan, and DI August. Design and evaluation of hybrid fault-detection systems. In *32nd Int. Symp. on Comp. Arch., Proc.*, pages 148–159, 2005. Madison, WI, Jun 04-08, 2005.

[14] P Bernardi, LMV Bolzani, M Rebaudengo, MS Reorda, FL Vargas, and M Violante. A new hybrid fault detection technique for systems-on-a-chip. *IEEE Trans. Comput.*, 55(2):185–198, Feb 2006.

[15] Felipe Restrepo-Calle, Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, F.R. Palomo, and M.A. Aguirre. Hardening development environment for embedded systems. 2010. In the 2nd HiPEAC Workshop on Design for Reliability (DFR'10) held in conjunction with The 5th Int. Conf. on High Performance and Embedded Architectures and Compilers. Pisa, Italy, Jan 25-27, 2010.

[16] 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 Transactions on Instrumentation and Measurement*, 58(5), May 2009.

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

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

[19] SK Reinhardt and SS Mukherjee. Transient fault detection via simultaneous multithreading. In *27th Int. Symp. on Computer Architecture*, pages 25–36, 2000. Vancuver, Canada, Jun 12-14, 2000.

[20] J Napoles, H Guzman, M Aguirre, J Tombs, F Munoz, V Baena, A Torralba, and L Franquelo. Radiation environment emulation for VLSI designs A low cost platform based on xilinx FPGAs. In *IEEE Int. Symp. on Industrial Electronics, ISIE 2007*, 2007.