UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática de Sistemas y Computadores

Detección concurrente de errores en el flujo de ejecución de un procesador

Tesis Doctoral

Francisco Rodríguez Ballester

Dirigida por:

Dr. D. Juan José Serrano Martín

Septiembre de 2015

A Susana, por su amor y paciencia

A mi padre, al que le habría gustado ver este trabajo terminado
Y a mi madre, que sonreirá al verlo

Agradecimientos

Quiero agradecer a mi familia y en especial a mi mujer Susana el amor, apoyo y comprensión que siempre me han brindado. Sentir ese respaldo incondicional es una bendición cuando te enfrentas a un trabajo de este calibre.

También quiero agradecer desde aquí el apoyo y solidaridad de todas las personas del Departamento de Informática de Sistemas y Computadores de la Universitat Politècnica de València que me han ayudado y animado durante el desarrollo de este trabajo de tesis. Son muchas las que, de una manera u otra, han aportado su granito de arena para que pudiera terminarlo. No quiero cometer el horrible pecado de dejarme a ninguno fuera de la que sería una larga lista, así que no voy a nombraros personalmente; espero que sepáis perdonarme este pecado venial.

No puedo sin embargo dejar de señalar a Toni Martí, colega y compañero de fatigas, investigaciones, cursos y demás embrollos, siempre dispuesto a arrimar el hombro.

Y, cómo no, mi mas sincera gratitud a la excepcional persona que es Juan José Serrano, colega, mentor y amigo, amén de director de esta tesis. Sin él no hubiera empezado. Y sin su contínuo apoyo y su ayuda en todos los momentos en los que la he necesitado definitivamente este trabajo no habria visto la luz.

Resumen

La incorporación de mecanismos de detección de errores es un elemento fundamental en el diseño de sistemas tolerantes a fallos en los que, en muchos casos, la detección de un error (ya sea transitorio o permanente) es el punto de partida que desencadena toda una serie de acciones o activación de elementos que persiguen alguno de estos objetivos: la continuación de las operaciones del sistema a pesar del error, la recuperación del mismo, la parada de sus operaciones llevando al sistema a un estado seguro, etc. Objetivos, en definitiva, que pretenden la mejora de las características de fiabilidad, seguridad y disponibilidad, entre otros, del sistema en cuestión.

Uno de estos elementos de detección de errores es un procesador de guardia; su trabajo consiste en monitorizar al procesador del sistema y comprobar que no se producen errores durante la ejecución del programa.

El principal inconveniente de las propuestas existentes a este respecto y que impiden una mayor difusión de su uso es la pérdida de prestaciones y el aumento de consumo de memoria que sufre el sistema monitorizado. El aumento en el consumo de memoria se debe a la adición al programa original de datos (denominados firmas) que contienen la información necesaria para permitir la detección de errores. Y la pérdida de prestaciones proviene del hecho de que, en general, se trata de que sea el propio procesador del sistema el que realice las operaciones necesarias (o, al menos, que recupere las firmas) para detectar posibles errores durante la ejecución.

En este trabajo se propone una nueva técnica de empotrado de firmas (técnica denominada ISIS — *Interleaved Signature Instruction Stream*) intercaladas dentro del espacio de la memoria del programa. Con esta técnica es un elemento separado del procesador del sistema (un procesador de guardia como tal) el que realiza las operaciones encaminadas a detectar los errores. A pesar de que las firmas se encuentran mezcladas con las instrucciones del programa que está ejecutando, y a diferencia de las propuestas previas, el procesador principal del sistema no se involucra ni en la recuperación de las firmas ni en las operaciones de cálculo correspondientes, lo que reduce la pérdida de prestaciones.

También se propone una novedosa técnica para que el procesador de guardia pueda verificar la integridad estructural del programa que monitoriza comprobando las direcciones de salto empleadas. Esta técnica de procesado de las direcciones de salto viene a resolver en gran medida el problema de la

comprobación de un salto a una nueva zona del programa cuando existen múltiples posibles destinos válidos. Este problema no tenía una solución adecuada hasta el momento, y aunque la propuesta que aquí se hace no consigue resolver todos los posibles escenarios de salto sí permite incorporar un buen números de ellos al conjunto de saltos verificables.

La propuesta teórica ISIS y sus mecanismos de detección de errores se complementan con la aportación de un sistema completo (procesador, procesador de guardia, memoria caché, etc.) basado en ISIS y que incorpora los mecanismos de detección que aquí se proponen. Se ha denominado HORUS a este sistema, y está desarrollado en lenguaje VHDL sintetizable, de manera que es posible no sólo simular el comportamiento del sistema ante la aparición de un fallo y analizar su evolución a partir de éste sino que también es posible programar un dispositivo lógico programable tipo FPGA para su inclusión en un sistema real.

Para programar el sistema HORUS se ha desarrollado en este trabajo una versión modificada del compilador `gcc` que incluye la generación de las firmas de referencia para el procesador de guardia como parte integral del proceso de creación del programa ejecutable (compilación, ensamblado y montaje) a partir de código fuente escrito en lenguaje C.

Finalmente, otro trabajo desarrollado en esta tesis es el desarrollo de FIASCO (*Fault Injection Aid Software COmponents*), un conjunto de *scripts* en lenguaje Tcl/Tk que permiten la inyección de un fallo durante la simulación de HORUS con el objetivo de estudiar su comportamiento y su capacidad para detectar los errores subsiguientes. Con FIASCO es posible lanzar cientos o miles de simulaciones en un entorno distribuido para reducir el tiempo necesario para obtener los datos de campañas de inyección a gran escala.

Los resultados demuestran que un sistema que utilice las técnicas que aquí se proponen es capaz de detectar errores durante la ejecución del programa con una mínima pérdida de prestaciones, y que la penalización en el consumo de memoria al usar un procesador de guardia es similar a de las propuestas previas.

Resum

La incorporació de mecanismes de detecció d'errors és un element fonamental en el disseny de sistemes tolerants a fallades. En aquests sistemes la detecció d'un error, tant transitori com permanent, sovint significa l'inici d'una sèrie d'accions o activació d'elements per assolir algun del objectius següents: mantenir les operacions del sistema malgrat l'error, la recuperació del sistema, aturar les operacions situant el sistema en un estat segur, etc. Aquests objectius pretenen, fonamentalment, millorar les característiques de fiabilitat, seguretat i disponibilitat del sistema.

El processador de guarda és un dels elements emprats per a la detecció d'errors. El seu treball consisteix en monitoritzar el processador del sistema i comprovar que no es produeixen error durant l'execució de les instruccions.

Els principals inconvenients de l'ús del processadors de guarda és la pèrdua de prestacions i l'increment de les necessitats de memòria del sistema que monitoritza, per la qual cossa la seva utilització no està molt generalitzada. L'increment del consum de memòria és conseqüència de la incorporació al programa original d'unes dades, anomenades signatures, que contenen la informació necessària per poder detectar els errors. Respecte de la pèrdua de prestacions, es deguda a que és el propi processador qui ha de realitzar les operacions necessàries per a detectar els errors durant l'execució de les instruccions.

En aquest treball es proposa una nova tècnica de encastat de signatures (tècnica anomenada ISIS — *Interleaved Signature Instruction Stream*) intercalantles en l'espai de memòria del programa. D'aquesta manera és possible que un element extern al processador realitze les operacions dirigides a detectar els errors, i al mateix temps permet que el processador execute el programa original sense tenir que processar les signatures, encara que aquestes es troben barrejades amb les instruccions del programa que s'està executant.

També es proposa en aquest treball una nova tècnica que permet al processador de guarda verificar la integritat estructural del programa en execució. Aquesta verificació permet resoldre el problema de com comprovar que, al executar el processador un salt a una nova zona del programa, el salt es realitza a una de les possibles destinacions que són vàlides. Fins el moment no hi havia una solució adequada per a aquest problema i encara que la tècnica presentada no resol tots el cassos possibles, sí afegeix un bon nombre de salts al conjunt de salts verificables.

Les tècniques presentades es reforcen amb l'aportació d'un sistema complet (processador, processador de guarda, memòria cache, etc.) basat en ISIS i que incorpora els mecanismes de detecció que es proposen en aquest treball. A aquest sistema se li ha donat el nom de HORUS, i està desenvolupat en llenguatge VHDL sintetitzable, la qual cosa permet no tan sols simular el seu comportament davant la aparició d'un error i analitzar la seva evolució, sinó també programar-lo en un dispositiu FPGA per incloure'l en un sistema real.

Per poder programar el sistema HORUS s'ha desenvolupat una versió modificada del compilador `gcc`. Aquesta versió del compilador inclou la generació de les signatures de referència per al processador de guarda com part del procés de creació del programa executable (compilació, assemblat i enllaçat) des del codi font en llenguatge C.

Finalment en aquesta tesis s'ha desenvolupat un altre treball anomenat FIASCO (*Fault Injection Aid Software COmponents*), un conjunt d'scripts en llenguatge Tcl/Tk que permeten injectar fallades durant la simulació del funcionament d'HORUS per estudiar la seua capacitat de detectar els errors i el seu comportament posterior. Amb FIASCO és possible llançar centenars o milers de simulacions en entorns distribuïts per reduir el temps necessari per obtenir les dades d'una campanya d'injecció de fallades de grans proporcions.

Els resultats obtinguts demostren que un sistema que utilitza les tècniques descrites és capaç de detectar errors durant l'execució del programa amb una pèrdua mínima de prestacions, i amb un requeriments de memòria similars als de les propostes anteriors.

Abstract

Incorporating error detection mechanisms is a key element in the design of fault tolerant systems. For many of those systems the detection of an error (whether temporary or permanent) triggers a bunch of actions or activation of elements pursuing any of these objectives: continuation of the system operation despite the error, system recovery, system stop into a safe state, etc. Objectives ultimately intended to improve the characteristics of reliability, security, and availability, among others, of the system in question.

One of these error detection elements is a watchdog processor; it is responsible to monitor the system processor and check that no errors occur during the program execution.

The main drawback of the existing proposals in this regard and that prevents a more widespread use of them is the loss of performance and the increased memory consumption suffered by the monitored system. The memory consumption increase is due to the addition to the original program of some data (called signatures) containing the required information to enable the detection of errors. And the performance loss comes from the fact that it is generally the system processor itself which perform the operations (or, at least, fetch the signatures) needed for the error detection mechanisms.

In this PhD a new technique to embed signatures is proposed. The technique is called ISIS — Interleaved Signature Instruction Stream — and it embeds the watchdog signatures interspersed with the original program instructions in the memory. With this technique it is a separate element of the system processor (a watchdog processor as such) who carries out the operations to detect errors. Although signatures are mixed with program instructions, and unlike previous proposals, the main system processor is not involved neither in the recovery of these signatures from memory nor in the corresponding calculations, reducing the performance loss.

A novel technique is also proposed that enables the watchdog processor verification of the structural integrity of the monitored program checking the jump addresses used. This jump address processing technique comes to largely solve the problem of verifying a jump to a new program area when there are multiple possible valid destinations of the jump. This problem did not have an adequate solution so far, and although the proposal made here can not solve every possible jump scenario it enables the inclusion of a large number of them into the set verifiable jumps.

The theoretical ISIS proposal and its error detection mechanisms are complemented by the contribution of a complete system (processor, watchdog processor, cache memory, etc.) based on ISIS which incorporates the detection mechanisms proposed here. This system has been called HORUS, and is developed in the synthesizable subset of the VHDL language, so it is possible not only to simulate the behavior of the system at the occurrence of a fault and analyze its evolution from it but it is also possible to program a programmable logic device like an FPGA for its inclusion in a real system.

To program the HORUS system in this PhD a modified version of the `gcc` compiler has been developed which includes the generation of signatures for the watchdog processor as an integral part of the process to create the executable program (compilation, assembly, and link) from a source code written in the C language.

Finally, another work developed in this PhD is the development of FIASCO (Fault Injection Aid Software Components), a set of scripts using the Tcl/Tk language that allow the injection of a fault during the simulation of HORUS in order to study its behavior and its ability to detect subsequent errors. With FIASCO it is possible to perform hundreds or thousands of simulations in a distributed system environment to reduce the time required to collect the data from large-scale injection campaigns.

Results show that a system using the techniques proposed here is able to detect errors during the execution of a program with a minimum loss of performance, and that the penalty in memory consumption when using a watchdog processor is similar to previous proposals.

Tesis por compendio de publicaciones

La tesis es un trabajo de creación inédita, una investigación rigurosa, un trabajo de producción científica, marco inicial de la especialidad de un investigador [1]. Constituyéndose en una fuente de información que refleja el logro en su propio campo del saber, estando directamente relacionado con la búsqueda y la transmisión del conocimiento a través de la información documentada, donde es de gran importancia la recopilación y el análisis de datos para el origen de la producción científica.

Con el ritmo actual de la transmisión del conocimiento, las Universidades y las Instituciones Profesionales y de Investigación de alto nivel han aceptado el papel de preparar a los futuros científicos y conceder el grado de Doctor a aquellos que demuestren ser capacitados para llevar a cabo investigaciones de alta calidad (Nascimento, 2000) y cuya producción científica sea una práctica habitual de la publicación de los resultados de las búsquedas que van surgiendo durante el desarrollo de las tesis doctorales, componiendo un trabajo original de investigación, que no siempre es totalmente inédito.

Basado en esta información, esta tesis doctoral se presenta en forma de un documento estructurado por compendio de artículos previamente publicados, guardando relación entre sí. Las publicaciones poseen calidad contrastada, en base al prestigio de la publicación en que han sido insertados.

La presentación de este documento, en esta aspecto concreto, busca estar de acuerdo con las normativas de estudio establecidas por el Programa de Doctorado en Arquitectura y Tecnología de los Sistemas Informáticos del Departamento de Informática de Sistemas y Computadores perteneciente a la Universitat Politècnica de València, de acuerdo con sus líneas de investigación, mediante la oportuna tramitación ofrecida dentro de su organización funcional y aprobación de la propuesta del Proyecto de Tesis por la dirección y la Comisión de Doctorado de esta Universidad.

Índice general

Agradecimientos III

Resumen IV

Resum VI

Abstract VIII

Tesis por compendio de publicaciones X

I Introducción 1

1. Introducción y objetivos de la tesis 2

1.1. Fundamentos 2

1.2. Motivación 5

1.3. Objetivos 6

1.4. Organización de la memoria 10

2. ISIS: Propuesta de empotrado de firmas 13

2.1. Introducción 13
2.2. Análisis de las propuestas existentes 15
2.3. Propuesta 19
2.3.1. Descripción de la firma de referencia 22
2.3.2. Mecanismos de detección de errores 24
2.3.3. Modificaciones al procesador y a los programas 28
2.3.4. Tratamiento de los saltos 34
2.4. Soporte software 39
2.4.1. Estructura interna del gcc 40
2.4.2. Elementos internos de las `binutils` más relevantes . . 42
2.4.3. Inserción automática de las firmas de referencia 46
2.4.4. Uso práctico del compilador 48
2.5. Conclusiones 49

3. HORUS: Implementación de la técnica ISIS 52

3.1. Introducción 52
3.2. Banco de pruebas 53
3.2.1. Organización del banco de pruebas 56
3.3. Arquitectura del sistema 57

II Publicaciones 60

4. A Watchdog Processor Architecture with Minimal Performance Overhead 61

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

5. The HORUS Processor **74**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

6. Delivering Error Detection Capabilities into a Field Programmable Device: The HORUS Processor Case Study **84**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

7. A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream **92**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

8. Improving the Interleaved Signature Instruction Stream Technique **102**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

9. Improving the Interleaved Signature Instruction Stream Technique **112**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

10. Control Flow Error Checking with ISIS **119**

Francisco Rodríguez, Juan José Serrano

11. Reducing the VHDL-Based Fault Injection Simulation Time in a Distributed Environment **132**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

12. A Distributed Simulation Environment for Fault Injection Analysis on SoC Models **142**

Francisco Rodríguez, José Carlos Campelo, Juan José Serrano

III Conclusiones 148

13. Conclusiones 149

13.1. Introducción 149

13.2. Aportaciones 151

13.3. Conclusiones 154

13.4. Publicaciones directamente relacionadas con el trabajo de tesis 157

13.5. Trabajo futuro 159

Parte I:

Introducción

Capítulo 1

Introducción y objetivos de la tesis

If you steal from one author it's plagiarism; if you steal from many it's research.

Wilson Mizner

1.1. Fundamentos

Los sistemas de control industrial, englobando bajo este concepto a los sistemas de control de maquinaria de fabricación, de automatización de procesos en diversos ámbitos, automoción, ferrocarriles, aviónica, agricultura, medicina, etcétera, han sido objeto de una constante evolución en los últimos años.

Esta evolución ha sido motivada por diversas causas. Entre éstas, podemos aludir a la flexibilidad demandada, es decir, la posibilidad de poder adaptar rápidamente el sistema ante cambios en el proceso a controlar, la rapidez y precisión requerida, así como la integración de estos sistemas en otros de mayor nivel, incremento de las prestaciones, reducción de costes e incremento de la competitividad.

A estas necesidades, por otra parte, se unen los avances en el campo tecnológico: autómatas programables más complejos, ordenadores especialmente diseñados para entornos industriales, microcontroladores cada vez más potentes, procesadores digitales de señal e importantes avances en las redes de

comunicación. En este último punto cabe destacar la aparición y estandarización de redes específicamente desarrolladas para este tipo de entornos, denominadas buses de campo o redes de área local industrial. Éstas han sido uno de los factores más importantes para favorecer el desarrollo de los sistemas distribuidos de control industrial.

Con estos avances ha sido posible abandonar la concepción clásica de los sistemas de control, típicamente centralizados, con multitud de conexiones punto a punto para obtener la información del proceso y generar las salidas, por sistemas distribuidos en los que las funciones a realizar se dividen entre una serie de dispositivos o nodos interconectados mediante una red de comunicación.

Este desarrollo ha posibilitado la aplicación de estos sistemas en multitud de aplicaciones, desde las más sencillas hasta las más complicadas y críticas. Esto ha hecho que, como importante requerimiento en estos sistemas, sea cada vez más frecuente, además de obtener una serie de prestaciones, demandar unos determinados índices de una serie de atributos como la fiabilidad, seguridad, disponibilidad, entre otros. Es decir, se exige que los sistemas sean de funcionamiento garantizado. Es en este punto donde el área de los sistemas tolerantes a fallos cobra una vital relevancia.

Se dice que un sistema es *tolerante a fallos* cuando es capaz de continuar su trabajo aunque se manifiesten errores permanentes, transitorios o intermitentes (errores para los cuales se han previsto mecanismos de tolerancia en la etapa de diseño del sistema). Por otra parte, se dice que un sistema tiene un *comportamiento seguro*, si una vez que se produce una avería y no se puede continuar el correcto funcionamiento, el sistema se detiene en un estado conocido que no causa ningún tipo de riesgo al proceso que controla.

La fiabilidad de los sistemas informáticos en general y de los sistemas industriales en particular, ha sido y es uno de los principales objetivos desde el punto de vista del diseño de estos sistemas. La introducción de los computadores en aplicaciones críticas desembocó en el avance definitivo de los sistemas tolerantes a fallos. Mejoras en la fiabilidad, seguridad, disponibilidad, confidencialidad, entre otros atributos de la garantía de funcionamiento, se han convertido en un requerimiento cada vez más importante en el diseño de estos sistemas.

Dentro del entorno industrial, el uso de sistemas distribuidos que controlen un proceso o aplicación que pueda entrañar un cierto riesgo, bien para el proceso que se intenta automatizar, o bien para los usuarios de los equipos que los

incorporen, hace necesario pensar en la garantía de funcionamiento que estos sistemas van a ser capaces de ofrecer. ¿Cuál va a ser la fiabilidad de estos sistemas?, ¿cómo advertir los errores para poder subsanarlos, o conseguir al menos que el sistema responda de forma segura?

Por tanto, hay que incorporar algunos mecanismos que nos permitan detectar los posibles errores que pueda sufrir un sistema. Estos errores podrán ser de tipo permanente, cuando algún componente del sistema se averíe o, de otro modo, de tipo intermitente o transitorio: cada vez más, tener sistemas que funcionan a frecuencias más altas y usando niveles de tensión más bajos (por tanto con una diferencia menor entre los valores lógicos) los hace ser más susceptibles a este tipo de fallos transitorios. Al mismo tiempo, el uso de muchos de estos sistemas en ambientes ruidosos, ambientes industriales, de automoción, transporte, entre otros, hace necesario incorporar, como hemos mencionado anteriormente, mecanismos para hacer que los sistemas funcionen correctamente ante la presencia de diversos errores.

El principal objetivo y también criterio de selección en el diseño de sistemas tolerantes a fallos es conseguir una alta garantía de funcionamiento. Sin embargo, otras variables importantes a tener en cuenta, y que a veces determinan la elección de un mecanismo frente a otros son:

- La pérdida de prestaciones que siempre implica la inclusión de mecanismos de tolerancia a fallos en un sistema. Esta caracterísitica hace que en la actualidad se aborde el problema del estudio conjunto de prestaciones y garantía de funcionamiento (*performability*), para comparar entre diferentes sistemas de tolerancia a fallos.

- El aumento de coste de un sistema tolerante a fallos frente a un sistema de funcionalidad similar pero que no tolere fallos; este coste es proporcional a la cantidad de redundancia añadida.

- La velocidad de respuesta de los mecanismos empleados, o la mayor o menor facilidad en su diseño.

Considerando como objetivo prioritario a la hora de diseñar un sistema tolerante a fallos la consecución de una alta garantía de funcionamiento, uno de los principales problemas a resolver es precisamente su validación. La validación basada en la evaluación de los modelos teóricos del sistema (basados en cadenas de Markov, redes de Petri o similares) en sus primeras fases de desarrollo no permite su conocimiento exacto, ya que deja por resolver dos grandes incógnitas:

- Los factores o coeficientes de cobertura en la detección o recuperación de los errores, que miden la bondad de los mecanismos de tolerancia a fallos introducidos.
- Los tiempos de latencia en la detección o recuperación de los errores, que miden la rapidez de los mecanismos de tolerancia a fallos introducidos.

Precisamente estas incógnitas son fundamentales a la hora de tomar decisiones acerca de un sistema tolerante a fallos. Por ello, la obtención exacta de estas incógnitas se basa en métodos experimentales. Además, debido sobre todo a la baja probabilidad de ocurrencia de fallos en estos sistemas, los experimentos no pueden basarse en la observación directa de un sistema tolerante a fallos en condiciones normales. EN definitiva, los experimentos se deben basar en la introducción (inyección) voluntaria y controlada de fallos en el sistema para observar su respuesta ante ellos.

1.2. Motivación

De forma general y desde el punto de vista de los sistemas distribuidos, la mejora de la garantía de funcionamiento se puede obtener mediante la aplicación de mejoras en tres niveles distintos:

1. A nivel de los nodos del sistema distribuido se pueden desarrollar arquitecturas tolerantes a fallos. Se intenta conseguir que el propio nodo se recupere de la avería y continue trabajando; si esto no es posible, el nodo debe ser capaz al menos de detectarla.
2. A nivel de sistema se puede mejorar la garantía de funcionamiento mediante la inclusión de estrategias de cooperación entre los distintos nodos para recuperar el sistema o para continuar funcionando, aunque sea con funcionalidades o prestaciones reducidas.
3. A nivel de red. La aplicación de niveles de red con características de tolerancia a fallos mejorará la garantía de funcionamiento, dado que se dispone de diferentes medios físicos para hacer llegar la información de una parte a otra del sistema.

El trabajo de esta tesis doctoral se centra en el primero de los niveles; concretamente en el diseño de mecanismos de detección de fallos en un sistema monoprocesador. El objetivo principal no es obtener mecanismos de enmascaramiento de fallos, que permiten que el sistema continue funcionando a pesar del fallo observado gracias a una enorme redundancia espacial o temporal, sino el de conseguir una alta tasa de detección que permita la utilización de los mecanismos de cooperación mencionados anteriormente para que el sistema global (el sistema distribuido, como una entidad) siga funcionando.

Esto no significa que otros mecanismos de tolerancia a fallos aplicados a esta misma clase de sistemas sea inútil o ineficaz, sino todo lo contrario, dado que los mecanismos de detección de errores aquí desarrollados pueden ser el complemento ideal para otro tipo de soluciones más enfocadas hacia la redundancia, tanto espacial como temporal.

1.3. Objetivos

Desde hace tiempo se está trabajando en el área de la detección de errores en los sistemas computadores, y se han creado técnicas para la detección concurrente de errores en el flujo de ejecución de un procesador con el fin de dotar al sistema de un cierto nivel de integridad en lo que a ejecución de instrucciones se refiere.

El problema de los errores durante la ejecución de instrucciones es tan apremiante (no hablamos aquí de los errores en los cálculos realizados sobre los datos, problema tan crítico como el que aquí se trata) que se han desarrollado técnicas puramente software para dotar a procesadores de propósito general de cierta capacidad de detección de los mismos.

El uso de estas técnicas software provoca una enorme sobrecarga para el procesador, con la consiguiente pérdida de prestaciones. Con el avance de la tecnología y la capacidad de integración estas técnicas han sido superadas gracias a la proposición de innovaciones a nivel de diseño de la circuitería del procesador o del sistema en el que se integra, cada día incorporadas de forma más sencilla y eficiente en el desarrollo de nuevos productos.

Sin embargo, las técnicas con soporte hardware están mayoritariamente basadas en el desarrollo y utilización de nuevas instrucciones, específicamente diseñadas para ayudar en la detección de errores durante la ejecución.

Disponer de una circuitería específica para verificar el funcionamiento del sistema ha permitido ampliar el campo de los errores detectables, incorporando no sólo la verificación estructural del programa (que es a lo más que llegan las técnicas software) sino también la verificación de cada una de las instrucciones que componen el programa.

Sin embargo, hacer partícipe al procesador de la detección de sus propios fallos de ejecución mediante la incorporación de nuevas instrucciones ha producido (en las propuestas planteadas) algunos o todos de los siguientes inconvenientes:

1. El procesador al que se aplica no es un procesador de propósito general, sino un procesador con una tarea muy específica. El diseño de dicho procesador no permite la inclusión de mecanismos (interrupciones, excepciones) absolutamente necesarios para cualquier sistema de propósito general.

2. El programador del sistema debe tener unos conocimientos muy específicos sobre detección de errores, para poder insertar correctamente las instrucciones correspondientes. Esto es mucho más grave de lo que pueda parecer en un principio, pues la necesaria especialización de estos programadores no permite la adecuada difusión del uso de técnicas de detección de errores. Por no hablar de que la incorporación del programador a la cadena de elementos que deben trabajar de forma armoniosa para la consecución del objetivo final (la detección de errores) lleva aparejada de forma inherente la posibilidad de que el sistema sufra fallos producidos por la intervención humana en dicha cadena. Podríamos considerar a éstos últimos un caso especial de los llamados *errores de diseño*.

3. El procesador resultante ya no permite la ejecución de código binario de la arquitectura de la que originalmente proviene. Esto también produce un importante rechazo a la difusión de las técnicas de detección de fallos, pues en lugar de permitir una incorporación gradual y paulatina en los sistemas existentes se fuerza a revisar todo el código ya implantado.

Y al mismo tiempo, dichas propuestas han dejado algunos o todos de los siguientes puntos sin resolver:

1. La pérdida de prestaciones y el incremento en el consumo de memoria al usar una técnica hardware es del mismo orden de magnitud que

la provocada con técnicas software. Es cierto que en este caso lleva aparejado un aumento en la cantidad de errores detectables, pero el coste es aún demasiado notable.

2. En la verificación de la integridad estructural del programa (la verificación del flujo de ejecución) no están contemplados todos los casos (estructuras de control, arquitectura del programa) que se pueden presentar en un software genérico, lo que supone abrir una ventana de incertidumbre (más o menos importante en función del uso que de estos casos no tratados se hace en el programa a verificar) sobre la integridad estructural del mismo.

El presente trabajo pretende dar un paso hacia la resolución de los inconvenientes anteriormente mencionados y reducir o minimizar al menos el impacto de los puntos sin resolver dejados como tema abierto de las propuestas previas.

En particular, esta tesis se centra en la proposición de una nueva formulación para la inserción de firmas en el flujo de ejecución de un procesador de propósito general que permita reducir la pérdida de prestaciones sin mermar la capacidad (la *cobertura*) de detección de errores ni empeorar el tiempo necesario para la detección (lo que se denomina *tiempo de latencia*).

Al mismo tiempo dicha proposición incluirá nuevos mecanismos de detección de errores con lo que se podrá reducir el número de escenarios no tratados.

Para comprobar de forma práctica la factibilidad de dicha propuesta, se diseñará un procesador basado en una arquitectura ampliamente conocida y utilizada en el mercado para incorporar los mecanismos propuestos. La implementación práctica demostrará no sólo la viabilidad de la propuesta, sino también la posibilidad de su aplicación a un sistema de propósito general en el que puedan convivir tareas a las que se les ha incorporado la información necesaria para los mecanismos de detección de errores y código binario estándar de la arquitectura base del procesador (la tantas veces nombrada *compatibilidad* con sistemas previos).

Amén de que la alteración de las tareas mencionada un poco más arriba no ha de suponer ningún conocimiento específico del programador más allá de incluir un parámetro más en el momento de compilar el código fuente; todo el trabajo será realizado de forma automática por el compilador y las herramientas sobre las que se apoya.

Para concretar, los objetivos específicos de esta tesis son los siguientes:

- Plantear una nueva propuesta que tenga en cuenta los factores limitadores de las propuestas previas, para incluir mecanismos de mejora.
- Identificar una arquitectura sobradamente conocida sobre la que llevar a la práctica la propuesta anterior. Plantear la arquitectura del sistema resultante.
- Demostrar la factibilidad de la arquitectura anterior, desarrollando un modelo en un lenguaje de descripción de hardware que permita su posterior síntesis sobre un dispositivo lógico programable. El hecho de disponer de un modelo sintetizable es lo que permite garantizar que la arquitectura es viable.
- Modificar las herramientas de desarrollo de software para dicha arquitectura de modo que permitan la incorporación, de la forma más transparente posible para el programador, de los mecanismos de detección de errores implementados.
- Demostrar mediante simulación o prueba experimental que los mecanismos de detección funcionan. No se trata de obtener una caracterización de los mismos, lo que exigiría un enorme trabajo adicional, aunque evidentemente ésta se dejará planteada como una de las líneas abiertas como continuación natural de este trabajo. Lo que se pretende demostrar en este caso es que la incorporación de los mecanismos de detección de errores al procesador original no es un mero artefacto cosmético, sino que dichos mecanismos están en disposición de llevar a cabo su tarea; esto es, permiten la detección de fallos en la ejecución del procesador de forma concurrente a la ejecución propiamente dicha.

  Otra cosa es llegar a cuantificar las características del sistema, en lo que a detección de fallos se refiere, tras la incorporación de dichos mecanismos. Esta caracterización incluiría, entre otros, los siguientes parámetros: i) el porcentaje de los fallos que, aun ocurriendo en el sistema, no resultan en una avería o mal funcionamiento del mismo, ii) la *cobertura de detección* o procentaje de fallos que, produciendo un error en el sistema, son efectivamente detectados (puesto que ningún mecanismo de detección es perfecto, algunos errores no serán detectados), iii) el impacto que supone, para las averías del sistema, la incorporación de nueva circuitería en principio diseñada para la detección de errores, etc.

- Cuantificar la penalización en la que se incurre por dotar al sistema de dichos mecanismos: el incremento en el consumo de la memoria necesaria y la degradación de las prestaciones obtenidas son en este punto los valores a obtener. Para este objetivo particular será necesario hacer un análisis de las causas que producen dicha penalización, de manera que los experimentos a realizar estén claramente encaminados a producir sobre el sistema el máximo estrés posible.

1.4. Organización de la memoria

Este documento se ha desarrollado en tres partes: Una primera parte escrita específicamente para la redacción de este documento en la que se introducen las ideas y conceptos fundamentales del trabajo desarrollado; una segunda parte en la que se incorporan las publicaciones asociadas a esta tesis doctoral en capítulos individuales adaptados al formato de este documento; y una parte final también inédita en la que se presentan las conclusiones y se formulan posibles caminos para la continuación del trabajo que aquí se presenta.

Esta estructura es la que se especifica en la Normativa de los Estudios de Doctorado de la Universitat Politècnica de València aprobada por el Consejo de Gobierno en su sesión de 15 de diciembre de 2011 (publicado en el Bulletí Oficial de la Universitat Politècnica de València nº 54) y modificada por acuerdo de la Comisión de Doctorado el 9 de abril de 2013 y aprobada en Consejo de Gobierno el 25 de abril de 2013 para una tesis doctoral por compendio de publicaciones.

En este capítulo (Capítulo 1) se presenta el entorno en el que se enmarca el trabajo de esta tesis y se desgranan los objetivos que se persiguen con su desarrollo.

El Capítulo 2 se centra en describir la nueva propuesta de generación e inserción automática de firmas a lo largo del código ejecutable de una aplicación que permite incorporar un mecanismo de detección concurrente de errores (más concretamente un *procesador de guardia*) sin que el usuario (el programador del sistema en este caso) necesite ningún conocimiento específico. A esta propuesta se le ha dado el nombre de una antigua deidad egipcia, ISIS, coincidiendo "casualmente" con el acrónimo de *Interleaved Signature Instruction Stream*.

El Capítulo 3 desarrolla la propuesta teórica del capítulo anterior, propo-

niendo las modificaciones mínimas necesarias a un procesador RISC basado en la arquitectura MIPS para implementar de forma práctica las propuestas planteadas con ISIS, así como las modificaciones al sistema de desarrollo de software, herramientas basadas en el compilador `gcc`, con una amplia base de usuarios tanto en los sistemas de propósito general como en los entornos específicos de los sistemas empotrados y con soporte para la arquitectura de procesadores MIPS.

No sólo se han incorporado en este procesador las propuestas de ISIS; además se ha dotado al procesador resultante (denominado HORUS) de la capacidad de alternar, en tiempo de ejecución, entre tareas que incorporan las técnicas de ISIS y tareas compatibles a nivel binario con la arquitectura MIPS original.

Este procesador se ha desarrollado utilizando el lenguaje de descripción de hardware VHDL, con el fin de poder pasar del modelo de simulación a un dispositivo lógico programable real tras el correspondiente proceso (automático) de síntesis.

Además del procesador se describe en este capítulo el conjunto de elementos adyacentes al mismo y que también han sido desarrollados para disponer de un sistema completo sobre el que poder realizar posteriores experimentos. Estos elementos son:

- El procesador de guardia que monitoriza y verifica el funcionamiento del procesador principal. Este procesador basa dicha verificación en los valores insertados en el programa a ejecutar en el momento de la compilación/enlazado (valores que se denominan *firmas*).
- Una memoria caché con dos puertos de acceso para permitir la lectura de una instrucción por el procesador principal y de una firma por el procesador de guardia de forma simultánea.
- El conjunto de elementos necesarios para construir el bus multimaestro de altas prestaciones AMBA-AHB siguiendo la especificación desarrollada por ARM. Este bus multimaestro va a permitirnos conectar el procesador y su caché de instrucciones con un conjunto de periféricos y, lo que es más importante desde el punto de vista de este trabajo, con la memoria del sistema.
- La interfaz con el exterior del dispositivo programable para disponer de una memoria ROM y RAM de tipo estática que conforma la memoria del sistema en la que se almacenan instrucciones y datos.

Para la realización de los experimentos realizados sobre el modelo VHDL del sistema que nos permitan demostrar la funcionalidad de los mecanismos de detección de fallos se ha desarrollado una herramienta específica denominada FIASCO (acrónimo de *Fault-Injection Aid Software Components*). Con esta herramienta (conjunto de *scripts* sería más correcto) se han podido inyectar fallos sobre el modelo VHDL del sistema, a fin de verificar que el fallo, una vez ha dado lugar a una avería, es detectado por el procesador de guardia.

FIASCO está íntimamente ligado con el simulador de VHDL utilizado a lo largo de esta tesis, Modelsim, de Mentor Graphics. Con el conjunto FIASCO+modelsim es posible diseñar campañas de inyección de fallos sobre el modelo del sistema HORUS, lanzando posteriormente la ejecución de cientos o miles de simulaciones a un conjunto de máquinas en un entorno distribuido. Los resultados son recogidos de forma automática y procesados para determinar si se ha producido una avería, si el procesador de guardia ha detectado el fallo, etc.

También se han obtenido datos para la evaluación de prestaciones del procesador principal:

- Por un lado, el análisis del código binario resultante tras la inserción de las firmas y su comparación con el código binario original permite determinar el incremento en el uso de la memoria.

- Por otro lado, la simulación del sistema con una carga determinada permite, comparando el tiempo de ejecución del sistema cuando la carga (el programa) incorpora firmas para el procesador de guardia respecto del caso original en el que no existen dichas firmas y una vez se ha inhibido el funcionamiento del procesador de guardia (para impedir que interfiera en el funcionamiento de la memoria caché de instrucciones), determinar la pérdida de prestaciones relativa por la inclusión del procesador de guardia en el sistema.

Finalmente, un resumen de las aportaciones de este trabajo, las publicaciones a las que ha dado lugar, así como la descripción de algunas de las principales líneas de trabajo abiertas están plasmadas en las conclusiones de esta memoria, recogidas en el Capítulo 13.

Capítulo 2

ISIS: Propuesta de empotrado de firmas

> *Yet, it is alarming to observe that the explosive growth of complexity, speed, and performance of single-chip processors has not been paralleled by the inclusion of more on-chip error detection and recovery features..*
>
> A. Avizienis [2]

2.1. Introducción

En el "modelo para el futuro" que Avizienis describe en [2] queda patente la urgente necesidad de incorporar mecanismos de tolerancia a fallos en los sistemas de computación que usamos a diario: *"Yet, it is alarming to observe that the explosive growth of complexity, speed, and performance of single-chip processors has not been paralleled by the inclusion of more on-chip error detection and recovery features"*.

Como primer paso para esta incorporación es de una importancia fundamental disponer de mecanismos de detección de errores eficientes. Dado que la inmensa mayoría de los fallos son de naturaleza transitoria, el uso de mecanismos concurrentes de detección de errores es del máximo interés, dadas las características de alta cobertura y, simultáneamente, mínima latencia de

detección, características primordiales a la hora de permitir la recuperación del sistema ante la avería producida.

Y como los experimentos nos han demostrado [3, 4, 5, 6], un alto porcentaje de los errores no sobreescritos acaba produciendo un error en el flujo de ejecución del procesador.

Al respecto de la incoporación de técnicas de tolerancia a fallos en el sector del consumidor medio, Siewiorek en [7] es tajante al afirmar que *"To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users"*.

Esto es, una técnica de tolerancia a fallos sólo puede ser considerada transparente si, como resultado de su aplicación, el sistema sufre un impacto mínimo en sus prestaciones, consumo de memoria o velocidad de procesamiento. No nos hemos de olvidar aquí del elemento humano; caso de que dicha técnica requiera de conocimientos altamente especializados en el área de la tolerancia a fallos podemos concluir, de manera casi automática, que dicha técnica no será utilizada en absoluto o, lo que resultaría aún peor, que será mal utilizada creando una falsa confianza en el sistema resultante.

También hemos de tener en cuenta que este trabajo, aunque guiado por necesidades de prestaciones y facilidad de uso, está dirigido al diseñador de sistemas empotrados. Este es nuestro "usuario final", el elemento humano al que antes aludíamos, y no al consumidor que, de hecho, hace uso de estos sistemas.

Es bajo estas premisas que se desarrolla la propuesta que se presenta en este capítulo. Se trata de una nueva técnica para empotrar las firmas de referencia que ha de usar un procesador de guardia, utilizando como líneas maestras: en primer lugar, la búsqueda del mínimo impacto en las prestaciones del sistema resultante y, en segundo lugar, la inclusión del máximo número posible de mecanismos de detección de errores.

Para dar forma a esta propuesta, en la sección siguiente se recogen las conclusiones más importantes obtenidas tras una revisión del estado de la cuestión y se analizan los puntos débiles de las propuestas disponibles en la literatura de la materia. De este análisis se extraen las características fundamentales de la propuesta, plasmada en la sección 2.3. Junto a la técnica de empotrado de firmas se propone una firma de referencia con un conjunto de mecanismos de detección de errores; éstos son descritos en la sección 2.3.2. Para finalizar, en la sección 2.4 se describe el soporte software que resulta necesario para

permitir la generación automática de las firmas de referencia.

2.2. Análisis de las propuestas existentes

De las propuestas preexistentes eliminaremos directamente de nuestro análisis aquéllas puramente software por su alto coste en prestaciones y su prácticamente nula capacidad para detectar errores producidos por la alteración de las instrucciones a ejecutar.

Tampoco extenderemos dicho análisis a las propuestas que exijan del procesador en cuestión elementos que, habiendo sido incorporados en los computadores que usamos a diario, aún están lejos de extenderse a los procesadores de los sistemas empotrados sobre los que versa este trabajo: la ejecución de múltiples instrucciones por ciclo de reloj de los procesadores superescalares o la capacidad de *"multithreading"*, por citar algunos ejemplos.

Centrándonos pues en las propuestas con un importante soporte de hardware sobre procesadores escalares, los principales puntos débiles son, en nuestra opinión:

- La falta de soporte de mecanismos cruciales para procesadores de propósito general. Así, por ejemplo, en la propuesta para el controlador de comunicaciones TTP se elimina del procesador la capacidad para tratar excepciones e interrupciones. Es bien cierto que, para el propósito específico de dicho controlador de comunicaciones, la tarea puede muy bien resolverse sin estos mecanismos. Pero también es cierto que un procesador de propósito general sin ellos es, hoy por hoy, inconcebible.
- La falta de soporte software para automatizar el proceso de instrumentación de los programas que deben ser verificados durante la ejecución. Podemos traer a colación aquí, como buen ejemplo de lo mencionado, la propuesta del *Instruction Checker Module* que se hace en [8]. En ésta, es necesario incorporar una instrucción específica de CHECK inmediatamente antes de la instrucción a verificar. Aunque hay una vaga referencia a que la verificación debe hacerse sobre las partes críticas del software que el procesador debe ejecutar, no hay establecida una vía clara que el programador pudiera utilizar desde un lenguaje de alto nivel para conseguir tal fin. Y, a la hora de realizar los experimentos sobre un modelo de simulación, las instrucciones de CHECK son insertadas

on-the-fly cuando se detecta la ejecución de una de esas instrucciones críticas. La justificación de este método en lugar de incorporar las susodichas instrucciones al código original es, justamente, la falta de soporte software. También se utiliza la misma justificación en [9] para omitir los resultados de su procesador de guardia en los experimentos de inyección de fallos. Aunque, de forma puntual, esta justificación puede ser facilmente entendida y aceptada, lo cierto es que no deja de generar dudas sobre la viabilidad de dichas propuestas, que debería ser despejada por los autores. En cualquier caso, una nueva propuesta no debería tomar ese mismo camino, sino fijar de forma inequívoca cómo se ha de usar el software de apoyo, y cómo éste genera la información necesaria para que los programas sean instrumentados.

- La inclusión del cálculo y comprobación de las firmas en tiempo de ejecución en las unidades de ejecución del procesador. Esta inclusión puede, potencialmente, afectar negativamente a la frecuencia máxima de funcionamiento al incorporar nueva circuitería a las unidades de ejecución del procesador cuyo funcionamiento se pretende monitorizar. Si los circuitos dedicados a la detección de errores se introducen sobre la ruta de mayor coste en las etapas de cálculo, es posible que haya que reducir la frecuencia del sistema para cumplir los nuevos requisitos temporales. Es necesario dejar claro que esta penalización es potencial y que sólo tras cada implementación de la técnica en cuestión se puede determinar si dicha penalización es real; el problema es que, si para entonces se demuestra que dicha penalización existe, es demasiado tarde para corregirla.

- La inclusión de nuevas instrucciones para el procesador a verificar, de manera que sea el propio procesador el que realice, como parte inherente de la ejecución de un programa instrumentado, los cálculos de la firma en tiempo de ejecución, la comparación con la firma de referencia, etc. En este apartado podemos incluir aquellas propuestas que, dejando el cálculo y la comprobación de las firmas en tiempo de ejecución a módulos especializados separados del procesador, exigen que éste traiga las nuevas instrucciones de memoria y las "ejecute", inyectándolas en el pipeline como parte del programa de la aplicación. Sea como fuere, el coste en prestaciones es evidente si hacemos caso a lo indicado en [10] sobre la longitud de los bloques de instrucciones: entre 7 y 8 de media según los autores. Añádase a cada uno de estos bloques una instrucción de no operación (ninguna otra instrucción puede tener un impacto menor) y tendremos una idea aproximada sobre la pérdida de

prestaciones en la que se incurre al utilizar estas propuestas.

- La ubicación del procesador de guardia excesivamente lejos del procesador principal. En efecto, en algunas propuestas el procesador de guardia se conecta al bus de acceso a memoria, utilizando técnicas de *"snooping"* para recuperar las firmas. Sin embargo, no es el procesador principal el que está directamente conectado a este mismo bus, sino el controlador de la memoria caché de primer nivel. De esta manera, se pierde visibilidad respecto del flujo de ejecución del programa que se intenta verificar; para compensar esta pérdida, se modifica el controlador de la memoria caché para que emita, hacia el mencionado bus, las firmas que el procesador de guardia necesita observar. Se mantiene sin embargo una importante pérdida de visibilidad que impide al procesador de guardia verificar si las instrucciones ejecutadas por el procesador principal se han corrompido o no. Sólo puede, entonces, realizar una verificación de la estructura del programa; se pierde en este caso la ventaja del soporte hardware y se consigue la verificación, únicamente, de la integridad en la estructura del programa.

Aparte de estos puntos débiles observados en las propuestas que se pueden encontrar en la literatura de la materia, y que habrán de ser evitados en la medida de lo posible, existen algunos aspectos de importancia que no deben ser olvidados a la hora de realizar un nuevo planteamiento.

Uno de estos aspectos es la ubicación de las firmas en memoria. Existen dos tendencias a este respecto. La primera consiste en separar completamente las firmas del programa que se pretende verificar, utilizando memorias separadas o, al menos, espacios distintos de la memoria del sistema. La segunda intercala las firmas de referencia del procesador de guardia entre las instrucciones del programa que se pretende verificar.

En el primer caso, el objetivo es evitar las interferencias que las firmas tendrían sobre la ejecución del programa si éstas se intercalaran entre las instrucciones del propio programa. Se crea entonces un nuevo problema, ¿cómo se asocia una firma al bloque de instrucciones del que ha de servir de referencia? Este problema, en general, se resuelve haciendo que el procesador de guardia ejecute un programa con un mínimo juego de instrucciones que le permita, al menos, mimetizar la estructura del programa del procesador principal.

En el segundo caso, el objetivo es justamente eliminar la sobrecarga que supone hacer que el procesador de guardia tenga que ejecutar un programa con

la misma estructura que el programa principal. En efecto, si la firma "acompaña" de algún modo al bloque de instrucciones, el procesador de guardia no necesita realizar ningún tipo de salto cuando el procesador principal ejecute una instrucción de salto: allá donde el procesador principal continúe la ejecución el procesador de guardia encontrará la firma de referencia asociada a las nuevas instrucciones.

Un segundo aspecto a tener en cuenta es cómo gestionar los saltos entre bloques del procesador principal. En la inmensa mayoría de las propuestas, los únicos saltos que se tratan son los saltos condicionales e incondicionales con un único destino posible. Es decir, no se tienen en cuenta estructuras de salto generadas desde un lenguaje de alto nivel que produzcan saltos a múltiples posibles destinos. Una de las escasas excepciones a esta simplificación utiliza una lista de posibles destinos para un salto múltiple. Ya los propios autores identifican dos problemas prácticos: el primero, que la lista de saltos es finita, de manera que no todos los escenarios de salto están contemplados; el segundo es la necesaria complejidad del procesador de guardia, que ha de ser capaz de realizar múltiples comparaciones entre la dirección de salto del procesador principal y las diferentes entradas de la lista para determinar si el salto es correcto o no, manteniendo al mismo tiempo la capacidad de analizar las instrucciones del procesador principal al ritmo al que éste las ejecuta.

Aunque, en principio, podría ser aceptable que, dadas las características de tolerancia a fallos del sistema que se está diseñando, se prohibiera a cambio el uso de este tipo de saltos, la realidad es que éstos son absolutamente comunes y necesarios para cualquier lenguaje de programación.

Piénsese que, cuando un programa termina la ejecución de una subrutina o función y se produce el retorno a la zona de programa llamante, dicho retorno es, desde el punto de vista del procesador de guardia, un salto, una ruptura de la secuencialidad. Puesto que una de las razones del uso de los procedimientos es la reutilización de código, es imposible asignar un único destino cuando se produce el retorno; sería tanto como forzar a que un procedimiento sólo pudiera ser ejecutado desde un único punto llamante.

Para permitir la utlización de funciones, algunos trabajos (véase a este respecto, por ejemplo, el trabajo descrito en [6]) proponen la instrumentación de los programas a monitorizar con instrucciones adicionales para manipular la pila del sistema y salvar así la necesaria información para que el procesador de guardia pueda verificar que el retorno es correcto.

Otros trabajos, como el descrito en [11] proponen, para el caso de saltos

con múltiples destinos, utilizar firmas de justificación y retrasar la verificación hasta que el programa, utilizando cualquiera de los múltiples caminos, alcance un punto común. Las firmas de justificación intercaladas hasta alcanzar dicho punto común consiguen que la firma en tiempo de ejecución sea la misma, independientemente del camino escogido por el procesador principal. Retrasar la verificación hasta dicho punto común tiene, como primer inconveniente, el notable incremento en la latencia de detección; también la cobertura se ve perjudicada, pues la secuencia de instrucciones puede llegar a ser bastante grande. Por último, resulta difícil imaginar cómo se podría aplicar esta técnica al caso de los retornos de procedimiento; esta técnica está más orientada a algunas estructuras de control de flujo de los lenguajes de alto nivel que se adaptan mejor a una ramificación múltiple y una relativamente pronta reunión en un punto común, como la sentencia condicional múltiple *switch* que se puede encontrar en el lenguaje C.

2.3. Propuesta

Para permitir la verificación de las instrucciones ejecutadas por el procesador principal (detectando así los denominados errores intra-bloque), las firmas a utilizar son calculadas utilizando las secuencias binarias de dichas instrucciones como datos de entrada a un LFSR (*Linear Feedback Shift Register*).

Estas firmas de referencia se ubican, en esta propuesta, junto al bloque secuencial asociado. Pero, a diferencia de la mayoría de las propuestas previas, la firma no utiliza el *delay slot* tras la instrucción de salto que termina, habitualmente, cada bloque. Por el contrario, la firma se ubica al comienzo del bloque.

Ubicar la firma precediendo al bloque asociado persigue un doble objetivo. Por un lado, podremos incluir un campo con la longitud exacta del bloque, en instrucciones. Esta longitud va a permitir que el procesador de guardia detecte todos los errores de inserción y borrado de salto, sin tener que esperar a que el bloque finalice (quizá incorrectamente) para realizar dicha verificación.

Por otro lado, esta ubicación nos va a permitir minimizar la pérdida de prestaciones. Para conseguirlo, bastará con que el procesador principal no tenga conocimiento de la existencia de las firmas. No utilizando una memoria separada, como en algunas propuestas previas, sino permitiendo la existencia de dos secuencias de ejecución intercaladas entre sí, pero al mismo tiempo

completamente separadas. Una de estas secuencias es la original del programa a verificar, la de las instrucciones del procesador principal. La otra contiene únicamente las firmas de referencia para que el procesador de guardia realice la detección de errores.

Esta idea, núcleo germinal de la propuesta de este trabajo, es la que da nombre a la técnica de empotrado de firmas que aquí se presenta: ISIS, acrónimo del inglés *Interleaved Signature Instruction Stream*, que podríamos traducir como *Secuencia de Instrucciones de Firma Intercalada*.

No basta, sin embargo, con ubicar la firma precediendo al bloque de instrucciones asociado para conseguir esta disociación. Se elimina solamente el problema cuando el procesador realiza un salto, pues éste se produce a la primera instrucción del bloque, evitando sencillamente toda referencia a la firma previa. Para conseguir la total independencia de las dos secuencias, aún es necesario determinar cómo el procesador principal puede "esquivar" la firma de referencia del bloque siguiente cuando, durante la ejecución de un salto condicional, por ejemplo, la condición no se cumple y el programa sigue la ejecución de forma secuencial. En este caso, entre el bloque que termina con la instrucción de salto condicional y el siguiente se intercala la firma de referencia de este último. Si permitimos al procesador principal continuar la ejecución secuencial sin más, la primera "instrucción" del bloque siguiente no será tal, sino su firma de referencia.

¿Cómo evitarlo? Con otra idea novedosa: modificar la *semántica* de las instrucciones de salto condicional del procesador. No se trata de cambiar la codificación de dichas instrucciones, sino de cómo se comporta el procesador durante su ejecución.

Para conseguir que el procesador evite la ejecución de la firma del bloque siguiente, se transforma el significado de la instrucción se salto condicional tradicional, de

$$CP = \begin{cases} \textit{destino} & \text{si la condición de salto se cumple} \\ \\ CP + 1 & \text{si la condición de salto no se cumple} \end{cases}$$

donde CP es el *Contador de Programa* y hace referencia a la dirección de memoria de la instrucción que se está ejecutando y *CP+1* es la posición de memoria siguiente a la actual; este comportamiento tradicional se transforma, como decíamos, en

$$CP = \begin{cases} destino & \text{si la condición de salto se cumple} \\ \\ CP + 2 & \text{si la condición de salto no se cumple} \end{cases}$$

creando de esta forma un *hueco* en la dirección $CP + 1$, justamente donde se ha de ubicar la firma del bloque siguiente que comienza, ahora, en la dirección $CP + 2$.

Téngase en cuenta que el hueco así creado, y aprovechado para insertar la firma de referencia del bloque siguiente, no supone coste de ejecución alguno para el procesador principal, ni se ha de modificar el proceso de compilación de los lenguajes de alto nivel para conseguirlo. Para el procesador principal dicho hueco sencillamente no existe, pues no realiza un salto como tal (un salto como resultado de la ejecución de una instrucción de salto incondicional o condicional con la condición de salto a valor cierto, se entiende).

Aún no resuelve este cambio semántico todos los problemas. Para una descripción completa de cómo se han disociado las dos secuencias de instrucciones, véase la sección 2.3.3. Sí es importante destacar, sin embargo, que no se requiere de la arquitectura sobre la que se pretenda aplicar la técnica ISIS de ninguna característica específica, ni tampoco es necesario modificar el juego de instrucciones.

El tratamiento que se da a los saltos con múltiples destinos, incluyendo los retornos de procedimiento, también es novedoso, y resuelve el problema de las múltiples verificaciones de forma sencilla y elegante. En el caso de que un salto tenga múltiples destinos, todos conocidos en el momento de generar las firmas, la verificación de que el salto se ha realizado correctamente se pospone hasta alcanzar el bloque destino. Allí, la firma de referencia del bloque contendrá la información necesaria para que el procesador de guardia pueda verificar que el *origen* del salto es el correcto. De esta forma, el procesador de guardia no necesita realizar un conjunto de comprobaciones de una lista más o menos larga, sino solamente una. Cada uno de los posibles destinos contiene la información que permite verificar el origien de dicho salto, repartiendo así por todo el programa la "lista" de destinos posibles. Al trasladar la verificación al bloque alcanzado se soluciona el problema de los saltos con múltiples destinos, aunque no completamente. Para una descripción exhaustiva de los posibles escenarios, y cuáles quedan aún sin resolver, véase la sección 2.3.4

	6 bits	3 bits	3 bits	4 bits	16 bits
Firma de referencia	Tipo	Dirección destino	Dirección origen	Longitud	Firma derivada

Figura 2.1: Codificación de la firma de referencia

2.3.1. Descripción de la firma de referencia

Apoyada sobre la técnica de empotrado de firmas se ha definido una firma de referencia con un conjunto de campos que pretende maximizar el número de mecanismos de detección de errores disponibles. La palabra que contiene la firma de referencia de un bloque de instrucciones contiene los siguientes campos (véase la figura 2.1 para una descripción de dicha palabra):

1. **Tipo de bloque**. La codificación del tipo de bloque se ha elegido de forma que no pueda ser confundida con una instrucción por el procesador principal. Esta codificación, siendo recomendable, no es imprescindible ni siquiera es siempre posible, pues depende del número de códigos de operación disponibles en el juego de instrucciones. En caso de poder realizarse, el sistema dispondrá de un mecanismo adicional de detección de errores para los saltos ejecutados por el procesador. Este mecanismo se ha denominado *inicio de bloque* y su descripción se puede consultar en la sección 2.3.2.

   Resulta obvio, sin embargo, que si la codificación del tipo de firma no puede diferenciarse completamente de la codificación de las instrucciones del procesador principal, el mecanismo de detección de errores mencionado no puede aplicarse.

   En cuaquier caso, la información que contiene este campo permite clasificar el bloque de instrucciones del procesador principal, y en función de dicha clasificación, realizar una verificación u otra del salto.

   Por un lado, los bloques se clasifican según el tipo de salto con el que finalizan en

   - El bloque acaba con un salto incondicional. En este caso, no se permite la ejecución secuencial al finalizar el bloque.
   - El bloque acaba con un salto condicional. Las verificaciones a realizar son análogas al caso anterior, con la diferencia de que se acepta que el procesador principal continúe la ejecución secuencial en lugar de saltar.

   y, de forma independiente, según los destinos en

- El bloque acaba con un salto simple. En este caso, y puesto que hay un único destino, la propia firma de referencia incorpora la información de verificación del salto en el campo *dirección destino*.
- El bloque acaba con un salto múltiple. La verificación del salto en este caso se pospone hasta alcanzar el bloque destino. Una vez allí, se utilizará el campo *dirección origen* de la firma asociada al bloque destino para dicha verificación.
- El bloque acaba con un salto no cubierto. Se trata aquí de dar cabida a los escenarios de saltos múltiples no cubiertos, y que quedan aún por resolver (ver sección 2.3.4). En lugar de prohibir dichos escenarios, se permite su existencia a costa de abrir una ventana de incertidumbre sobre el salto que el procesador realiza al finalizar uno de estos bloques.

Finalmente, en otra clasificación independiente de las anteriores, se caracteriza el bloque origen desde el que se alcanza el bloque actual:

- Si el bloque es alcanzado desde uno o más bloques de salto simples, la verificación del salto que proporciona el campo *Dirección origen* no ha de realizarse.
- Si el bloque es alcanzado desde un bloque de salto múltiple, la verificación del salto que proporciona el campo *Dirección origen* ha de realizarse para verificar que el salto es correcto.

2. **Dirección destino**. Los bits de este campo permiten verificar que el salto realizado por el procesador principal es correcto. La diferencia entre las direcciones de la instrucción de salto que finaliza el bloque en curso y la que resulta como destino del salto (en caso de un salto condicional, si es que la condición es cierta) se compacta utilizando un sencillo árbol de puertas lógicas xor que permite verificar que el salto que va a ejecutar el procesador para finalizar el bloque actual es correcto. La descripción del cálculo de los bits de este campo, y del siguiente (dirección origen), que se realiza exactamente de la misma forma, puede consultarse en la sección 2.3.2

3. **Dirección origen**. Los bits de este campo permiten verificar que el salto realizado por el procesador principal es correcto. Aunque el proceso de cálculo de los bits de este campo coincide con el campo anterior (dirección destino), en este caso el bloque actual es uno de los posibles destinos que se alcanza desde un bloque origen con múltiples destinos.

Así pues, la verificación que realiza el procesador de guardia con este campo se realiza *tras* el salto.

4. **Longitud**. El procesador de guardia verifica, cuando la última instrucción del bloque es retirada del pipeline, que el número de instrucciones ejecutadas desde el inicio del bloque es el correcto. La detección de que la instrucción corresponde a la última de un bloque la realiza el procesador de guardia comprobando que el procesador principal ha realizado un salto, una ruptura de secuencia.

5. **Firma derivada**. En este campo se almacena el resultado de un registro LFSR alimentado con la secuencia binaria de las instrucciones del bloque, que consiste en la firma asignada del bloque.

Ha de tenerse en cuenta que el tamaño de estos campos es orientativo, y que deben realizarse experimentos para caracterizar los mecanismos de detección de errores y determinar si el número de bits asignados a cada campo es el adecuado.

En particular, el campo *tipo de bloque* se ha escogido del mismo tamaño que el código de operación de las instrucciones de la arquitectura MIPS sobre la que se realiza el desarrollo práctico (véase a este respecto el capítulo 3). Por la misma razón, el tamaño total de la firma de referencia es de 32 bits, para que coincida con el tamaño de las instrucciones de la mencionada arquitectura.

2.3.2. Mecanismos de detección de errores

Con los campos de la firma de referencia, los mecanismos de detección de errrores que se incorporan al procesador de guardia son

1. **Inicio de bloque**. Aunque no siempre es posible, resulta muy recomendable codificar los tipos de firma de referencia de forma completamente diferenciada de las instrucciones del procesador principal. Esta codificación permite que el procesador de guardia incorpore un mecanismo de detección de errores de los saltos ejecutados por el procesador principal que se ha denominado inicio de bloque. El funcionamiento de este mecanismo de detección de errores es muy simple: en el caso de que el procesador salte a una dirección errónea y alcance una instrucción que no sea el inicio de un bloque de instrucciones, la posición de memoria

inmediatamente anterior a ésta no contendrá una firma de referencia para el procesador de guardia; y puesto que el procesador de guardia utiliza las direcciones del procesador principal como base para obtener la firma de referencia del bloque que éste está ejecutando, y gracias a la diferenciación total en la codificación, el procesador de guardia puede fácilmente detectar el error al encontrar un tipo de firma que no se corresponde con ninguno de los tipos conocidos. De forma análoga, también el procesador principal puede detectar directamente el error y activar una excepción de tipo *instrucción ilegal* en caso de que el salto erróneo alcance, como destino del mismo, una firma de referencia.

2. **Tipo de bloque**. Aunque el tipo de bloque, por sí mismo, no es un campo que permita hacer ninguna clase de verificación de la integridad del flujo de ejecución (a excepción de la indicación de salto condicional o incondicional), la secuencia de tipos entre las firmas de referencia de bloques consecutivos sí permite determinar si la estructura del programa ejecutado por el procesador es correcta, pues no todas las combinaciones son aceptables. Por ejemplo, si en un bloque la firma indica que la verificación del salto debe posponerse hasta que el procesador alcance el bloque destino, la firma asociada a dicho bloque destino debe indicar, necesariamente, que se trata de un bloque alcanzado desde un salto con múltiples destinos y que ha de realizarse la verificación de que el origen es correcto. De la misma forma, si un bloque acaba en un salto con un único destino, el bloque destino no puede indicar que se ha de realizar la verificación del bloque origen del salto.

3. **Dirección de bloque**. Bien sea verificando el bloque que se ha de alcanzar al finalizar el actual (utilizando el campo dirección destino) o el bloque origen desde el que el actual ha sido alcanzado (utilizando el campo dirección origen), la verificación de los saltos del procesador principal se realiza de la misma manera. Para calcular el valor de referencia se utiliza la diferencia entre las instrucciones origen y destino del salto, igual que el valor calculado por el procesador de guardia en tiempo de ejecución. Este valor es compactado en ambos casos mediante puertas lógicas xor para reducir la longitud (en bits) hasta acomodarse a la longitud del campo correspondiente.

4. **Longitud de bloque**. Este es uno de los mecanismos más sencillos. Se trata de un simple contador de instrucciones que se decrementa con cada instrucción ejecutada por el procesador principal. Si el contador llega a cero y no se produce una ruptura de secuencia, se ha producido

un error de borrado de salto. Si por el contrario la ruptura de secuencia se produce antes de que el contador llegue a cero, se ha producido un error de inserción de salto. Nótese que, para esta verificación, los saltos condicionales en los que la condición de salto no se cumple (y por tanto, no se toma el salto) se procesan exactamente igual que los saltos incondicionales. Dada la estructura del programa del procesador principal, y gracias al hueco creado en la secuencia de direcciones referenciadas por dicho procesador en el caso de un salto condicional no tomado, el procesador de guardia observa una ruptura de la secuencia de instrucciones independientemente del tipo de salto, o de si el salto se toma o no.

5. **Integridad de las instrucciones**. Con la firma derivada utilizada como referencia, y utilizando el mismo cálculo LFSR con las instrucciones que realmente ejecuta el procesador, el procesador de guardia puede verificar que las instrucciones se han mantenido íntegras durante su ejecución. Para conseguirlo, es necesario que las instrucciones sean analizadas por el procesador de guardia cuando éstas son retiradas del pipeline, y no cuando entran, pues durante las diferentes etapas del pipeline la instrucción se almacena en registros internos susceptibles de sufrir errores.

Hay que tener en cuenta que no son estos los únicos mecanismos de detección de errores de un sistema que implementa la técnica ISIS; son sólo los añadidos por el procesador de guardia. Pero la mayoría de procesadores incorporan uno o más mecanismos de verificación que fuerzan el tratamiento del error mediante una excepción, entre los que se puede citar

- Ejecución de instrucción ilegal.
- Búsqueda de instrucción en dirección no alineada en frontera de palabra.
- Búsqueda de instrucción en una dirección de página no mapeada en la MMU, o sin los adecuados privilegios de acceso.

Cálculo del campo de dirección

Como ya se ha mencionado anteriormente, los bits de referencia para la verificación de un salto se calculan mediante un árbol de puertas lógicas xor, a

partir de la distancia (en instrucciones) entre la instrucción de salto y la de destino del mismo.

Dicha diferencia, que denominaremos V, se compacta utilizando sus bits de forma alternada para conseguir una máxima cobertura, rellenando con ceros por la izquierda si resulta necesario. Algorítmicamente, podemos expresar cada uno de los bits de referencia para el salto como

para $i = 0$ hasta $K - 1$
$g_i = 0$
para $j = 0$ hasta $\lceil L/K \rceil$

$$g_i = \begin{cases} g_i \oplus V_{i+Kj} & \text{si } i + Kj < L \\ g_i \oplus 0 & \text{si } i + Kj \geq L \end{cases}$$

fin para
fin para

donde g_i es cada uno de los bits de referencia, K es el número total de dichos bits (e igual al tamaño del campo correspondiente en la firma) y L es la longitud, en bits, de la diferencia entre direcciones, V.

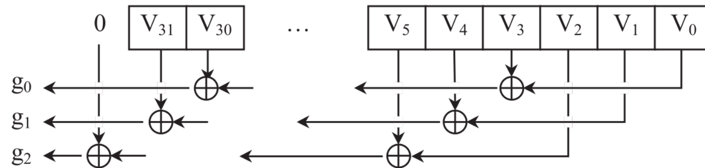

Figura 2.2: Árbol de puertas xor para la compactación del salto para un campo de 3 bits

En la figura 2.2 se puede observar un ejemplo de este cálculo aplicado al caso $K = 3, V = 32$. Esta expresión de los bits de referencia resalta dos cualidades del proceso de cálculo, a saber:

- Que una secuencia de bits contiguos todos ellos erróneos y de una longitud inferior a $2K$ que afecte al valor V será, con toda seguridad, detectada por el mecanismo. Es evidente que, si una secuencia de bits erróneos (del valor V) tiene una longitud inferior a $2K$, al menos uno de los bits calculados g_i está afectado por un único bit erróneo; y el

cálculo realizado con puertas xor garantiza que, con un único bit erróneo, el resultado diferirá, con toda seguridad, del de referencia (en ese bit particular).

- Que el número de bits de referencia, K, puede ser cambiado fácilmente buscando, por ejemplo, aumentar la cobertura ante errores múltiples, según lo expuesto anteriormente.

Hay que tener en cuenta que, aunque podemos partir de la suposición generalmente aceptada de la existencia de errores simples (de un único bit) como los más comunes con enorme diferencia, lo cierto es que la alteración de un único bit en uno de los operandos puede dar lugar, a través del proceso de cálculo del destino del salto que hace el procesador, a un error múltiple en el resultado. Y es a partir de este resultado que el procesador de guardia obtiene el valor V.

Se hace por ello necesario, en este caso particular de los saltos del procesador, poner un especial énfasis en la más que habitual existencia de errores de más de un bit, errores cuya detección resulta primordial en un sistema que pretenda ofrecer garantías de que el flujo de ejecución del procesador que se está monitorizando se mantiene íntegro.

2.3.3. Modificaciones al procesador y a los programas

En la sección 2.3 ya se han adelantado algunos detalles de cómo se ha conseguido mantener aisladas las dos secuencias de instrucciones que coexisten en un sistema que utilice la técnica ISIS. En esta sección se revisan estos detalles y se completan para conseguir una descripción completa de las modificaciones que es necesario realizar, tanto al procesador como al programa que éste ejecuta, para conseguir dicho aislamiento.

Se puede adelantar ya que algunas de estas modificaciones, en especial las que hacen referencia a la inserción de instrucciones en el programa original que el procesador principal ha de ejecutar, dan lugar a una de las causas de la pérdida de prestaciones que se analiza en el capítulo 13.

Aclaraciones previas

Antes de comenzar con la descripción detallada de estas modificaciones, es necesario realizar unas aclaraciones previas.

Sin restar generalidad a la propuesta, se describen modificaciones y soluciones a la arquitectura del procesador principal y a los programas que éste ejecuta, utilizando un ejemplo de implementación con las siguientes características:

- El tamaño de la firma de referencia coincide con el tamaño de una instrucción, y con una posición de memoria. Es decir, cuando se está ejecutando la instrucción en la dirección A y se desea hacer referencia a la instrucción siguiente, se indica como $A + 1$.
- La ejecución de una instrucción de salto **no** implica la ejecución de la instrucción que le sigue inmediatamente, se realice el salto o no, al contrario de lo que ocurre en la mayoría de los procesadores actuales (con pipeline). Esta instrucción, que ocuparía el denominado *delay slot*, es típica de procesadores con pipeline, ya que para cuando el procesador determina que el salto ha de realizarse, la siguiente instrucción (la que ocupa el *delay slot*) ya ha entrado en el pipeline.

Este ejemplo de implementación pretente que la lectura no sea en exceso farragosa (y, en este punto, el autor admite sugerencias de todo tipo para hacer el texto algo más ameno y algo menos espeso de lo que es).

Sin embargo, las modificaciones pueden aplicarse a otros sistemas completamente diferentes sin más que realizar las siguientes sustituciones:

1. Si denominamos W al espacio ocupado por una firma (en posiciones de memoria), cuando se mencionan direcciones de instrucción todas las referencias a $+1$ se han de sustituir por $+W$.
2. Si el sistema dispone de un pipeline que genera el denominado *delay slot*, cuando en el texto se menciona al registro CP referido a una instrucción de salto o de llamada a procedimiento debe entenderse referido a la instrucción del *delay slot* correspondiente.
3. En ese mismo caso, cuando en el texto se indica que, gracias a la modificación del comportamiento de determinadas instrucciones, se crea un hueco para alojar la firma de referencia en la posición de memoria

siguiente a la instrucción de salto, debe entenderse que se crea en la posición de memoria siguiente al *delay slot* correspondiente.

Modificaciones

En primer lugar, el ubicar la firma de referencia precediendo al bloque asociado permite garantizar que, cuando dicho bloque es alcanzado mediante un salto, la mencionada firma pase completamente desapercibida para el procesador principal.

Queda por resolver, entonces, cómo conseguir que el procesador principal no realice búsquedas de instrucción para recuperar firmas, cuando el bloque es alcanzado por otros medios. Se pueden distinguir 4 casos diferentes:

1. Aquellos bloques que son alcanzados por la simple ejecución secuencial cuando, al finalizar un bloque con una instrucción de salto condicional, la condición lógica correspondiente no se cumple y, por tanto, el procesador no salta.

2. Aquellos bloques que son alcanzados desde un salto, cuando dicho salto es el retorno de un procedimiento. Este caso requiere un tratamiento especial, dado que cuando se termina la ejecución del procedimiento y se ejecuta el retorno, la dirección utilizada para volver al programa llamante es calculada de forma automática por el procesador en el momento de la llamada al procedimiento. No se contemplan aquí los saltos realizados a direcciones calculadas en tiempo de ejecución cuando dicho cálculo está especificado de alguna manera como parte del programa que se está ejecutando y, por tanto, bajo el total control del programador.

3. Aquellos bloques que son alcanzados por la simple ejecución secuencial de instrucciones sin que medie instrucción de salto alguna entre ambos. Estos cambios de bloque, denominados *fall-through*, se producen cuando, dentro de una secuencia de instrucciones sin saltos, existe alguna instrucción que es referenciada desde una instrucción de salto. La instrucción referenciada se conoce como *branch-in* y su existencia exige que la secuencia de instrucciones se divida en dos bloques secuenciales, uno que acaba con la instrucción previa al *branch-in* y otro que comienza justamente con ésta.

4. Aquellos bloques que son artificialmente divididos en bloques de menor tamaño para ajustar su longitud al campo correspondiente de las firmas de referencia. Aunque la causa de la división es radicalmente diferente del caso anterior, su tratamiento puede muy bien asimilarse a éste, pues la situación de partida es la misma: es necesario dividir una secuencia de instrucciones y no existe una instrucción de salto, sólo se dispone de una indicación de por dónde hay que realizar dicha división.

Cada uno de estos casos requiere una atención específica, que se detalla a continuación.

Para el primer caso, como ya se ha mencionado al presentar la propuesta ISIS, se altera la semántica de las instrucciones de salto condicional para que se genere un hueco en la secuencia de direcciones referenciadas por el procesador, cuando éste ejecuta un salto condicional y decide que el salto no ha lugar.

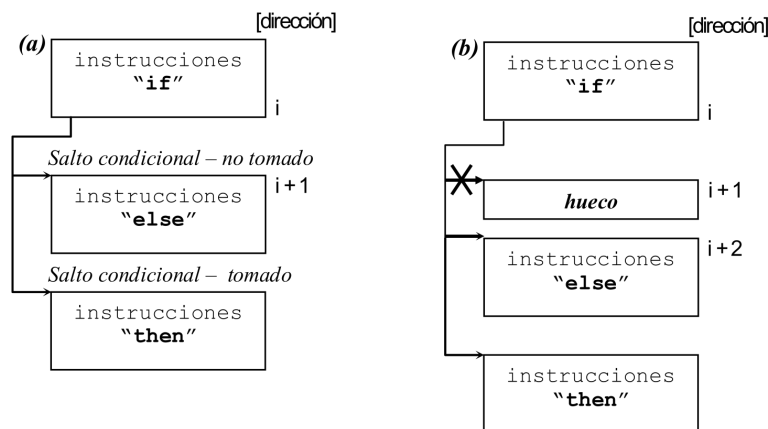

Figura 2.3: Ejemplo de alteración del comportamiento de una instrucción condicional: (a) antes de la modificación; (b) tras la modificación para crear un hueco en la dirección i+1

Este cambio implica modificar el comportamiento del procesador, de

$$CP = \begin{cases} destino & \text{si la condición de salto se cumple} \\ CP + 1 & \text{si la condición de salto no se cumple} \end{cases}$$

que es lo habitual, a

$$CP = \begin{cases} destino & \text{si la condición de salto se cumple} \\ \\ CP + 2 & \text{si la condición de salto no se cumple} \end{cases}$$

siendo CP el registro *Contador de Programa* que hace referencia a la dirección de memoria de la instrucción en curso (la de salto) y *CP+1* la posición de memoria siguiente a la actual.

Con este sencillo cambio se crea un *hueco*, una posición de memoria no referenciada, en la dirección $CP + 1$, justamente donde se ha de ubicar la firma del bloque siguiente que comienza, tras la modificación, en la dirección $CP + 2$.

En la figura 2.3a se puede observar un ejemplo típico de una instrucción de salto condicional (ubicada en la dirección i) utilizada como parte de una sentencia condicional de alto nivel `if-then-else`. En la figura 2.3b, tras aplicar la modificación mencionada, se crea un hueco en la dirección $i+1$, precediendo al bloque `else` de instrucciones que, con la modificación, comienza en la dirección $i + 2$.

Para el segundo caso, los retornos de procedimiento, se realiza otra modificación semántica al juego de instrucciones del procesador principal, en este caso al cálculo automático de la dirección de retorno cuando se realiza una llamada a procedimiento, pasando el comportamiento de las instrucciones de llamada a procedimiento de

$$\begin{aligned} &retorno = CP + 1 \\ &CP = destino \end{aligned}$$

a

$$\begin{aligned} &retorno = CP + 2 \\ &CP = destino \end{aligned}$$

Creando, otra vez, un hueco en $CP{+}1$, esta vez tras la instrucción de llamada a procedimiento. Hueco que, justamente, precede al bloque de instrucciones que se han de ejecutar cuando se realice el correspondiente retorno.

Para los dos últimos casos, en los que no hay instrucción de salto alguna que separe los dos bloques, la única solución consiste en modificar el programa

original para insertar, en el punto exacto donde se desea la división entre bloques, una instrucción de salto. Puesto que el salto se introduce únicamente para conseguir que el procesador "esquive" la firma que se ha de colocar entre ambos bloques, la instrucción de salto no obedece a la lógica original del programa, por lo que i) se utiliza una instrucción de salto incondicional a la dirección $CP + 2$, y ii) esta instrucción debe ser considerada como una necesaria sobrecarga del sistema, tanto en espacio de memoria como en tiempo de ejecución.

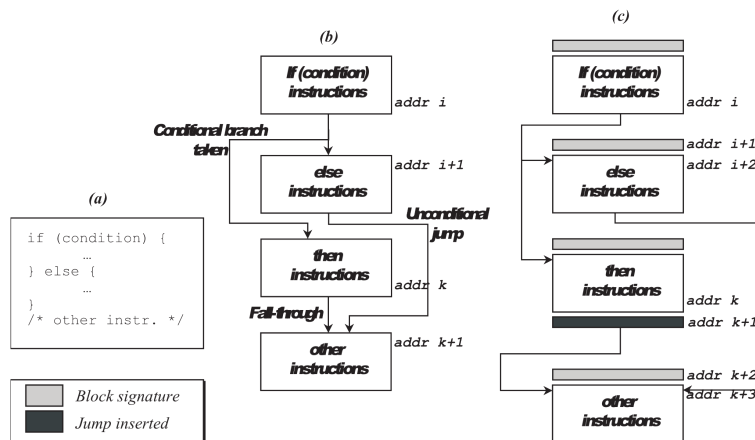

Figura 2.4: Ejemplo de una sentencia `if-then-else`: (a) alto nivel; (b) bloques antes de la inserción de firmas; (c) tras la inserción de firmas e instrucciones de salto

En la figura 2.4 se puede observar un ejemplo de estas moficaciones sobre una sencilla construcción de alto nivel `if-then-else`. En la figura 2.4b se observan los bloques secuenciales de instrucciones. Los bloques se mantienen separados para mejorar la legibilidad aunque las instrucciones de los mismos se suceden sin solución de continuidad. También se ha añadido a la derecha, una columna con algunas de las direcciones de instrucción, para evidenciar dicha continuidad.

En la figura 2.4c aparece el programa tras ser instrumentado; las líneas punteadas indican el comportamiento que tendría un procesador típico ante una instrucción de salto condicional. En la figura se observa, por un lado, que cada bloque es precedido de su correspondiente firma (en gris claro en la figura). Y por otro, que la transicion *fall-through* entre los dos últimos bloques se ha sustituido por la inserción forzada de una instrucción de salto incondicional (en gris oscuro en la figura, en la dirección $k + 1$) que hace que el procesador salte por encima de la firma de referencia (almacenada en la

dirección $k + 2$ en la figura) del último bloque, que comienza en la dirección $k + 3$ en la figura. Puesto que la instrucción es ejecutada por el procesador principal, es una instrucción más del bloque correspondiente.

2.3.4. Tratamiento de los saltos

Con ISIS se amplía el número de escenarios cubiertos por los mecanismos de detección de errores respecto de las propuestas previas. Sin embargo, no todos los saltos pueden ser verificados en tiempo de ejecución.

Como premisa inicial, para poder calcular los bits de referencia antes de la ejecución, es necesario que el destino o destinos de la instrucción de salto sea(n) conocido(s). Es esta una premisa que acompaña a todas las técnicas de verficación de la integridad estructural de un programa, incluída la técnica que aquí se propone.

Dando por supuesto que la premisa anterior se cumple, ISIS permite la verificación de algunos saltos con múltiples destinos mediante la verificación, utilizando la firma de referencia del bloque destino, de que el origen es correcto. Posponer la verificación hasta obtener la firma del bloque destino permite, de forma sencilla, eliminar el mayor obstáculo que impedía, tradicionalmente, verificar un conjunto de direcciones de salto de referencia.

Sin embargo, no todos los saltos pueden ser verificados. Para entender los posibles escenarios que se pueden dar en una aplicación, y de cuáles pueden ser verificados mediante los mecanismos incorporados con ISIS, es necesario crear una pequeña taxonomía de los tipos de salto:

1. Salto incondicional simple. Aquel salto que siempre se toma y que tiene un único destino. En este tipo de saltos se encuentran también las instrucciones de llamada a subrutina o salto a procedimiento.
2. Salto incondicional múltiple. Aquel salto que siempre se toma, y que tiene un conjunto finito y conocido de posibles destinos, de los cuales uno de ellos es seleccionado en tiempo de ejecución. En este caso se incluyen los retornos de procedimiento. Estrictamente hablando, y aunque son instrucciones de ruptura de la secuencia de ejecución, las instrucciones de retorno de procedimiento no están catalogadas como instrucciones de salto. Sin embargo, y para el desarrollo de mecanismos de verificación de un procesador de guardia, cualquier ruptura de secuencia puede asimilarse a un salto.

3. Salto condicional simple. Aquel salto que puede tomarse o no, pero que en caso de tomarse tiene un único destino posible y conocido. Aunque para los mecanismos de verificación de ISIS podría considerarse factible el caso de una ejecución condicional de procedimiento, esta posibilidad no se ha tenido en cuenta en el desarrollo por no encontrarse dentro del conjunto de instrucciones de la arquitectura MIPS, base de desarrollo de la parte práctica de este trabajo (véase el capítulo 3).

4. Salto condicional múltiple. Aquel salto condicional que, en caso de tomar el salto, tiene un conjunto finito y conocido de posibles destinos, uno de los cuales es seleccionado en tiempo de ejecución. Tampoco en este caso se ha considerado la posibilidad de una instrucción de retorno condicional, aunque, igual que en el caso anterior, si está soportada por una arquitectura y se dispone del soporte software correspondiente, los mecanismos de verificación incorporados en ISIS podrían tratar estos saltos de forma natural.

y otra taxonomía de los tipos de destinos que se pueden alcanzar con un salto:

1. *Branch-in* alcanzado desde una o más instrucciones de salto simples.
2. *Branch-in* alcanzado desde una única instrucción de salto múltiple.
3. *Branch-in* alcanzado desde una o más instrucciones de salto simples y una única instrucción de salto múltiple.
4. *Branch-in* alcanzado desde dos o más instrucciones de salto múltiple.
5. *Branch-in* alcanzado desde una o más instrucciones de salto simples y dos o más instrucciones de salto múltiple.

En esta última clasificación, y para simplificar, se ha omitido deliberadamente la mención al tipo de salto (condicional o incondicional) por tener el mismo tratamiento.

El primer caso es el único que, de forma mayoritaria, tenía tratamiento hasta ahora en las propuestas existentes en la literatura. El tratamiento con ISIS es similar a estas propuestas: en la firma de referencia del bloque origen (donde se encuentra la instrucción de salto) se incluyen los bits de referencia para verificar el salto desde el origen, en el campo *dirección destino*, y se indica la

verificación en el tipo; en la firma del bloque destino el contenido del campo *dirección origen* es indiferente, pues el tipo de firma indica que la verificación del origen no ha lugar.

El segundo caso se trata en ISIS retrasando la verificación hasta que el procesador de guardia dispone de la firma de referencia del bloque destino. Para ello, en la firma de referencia del bloque origen el contenido del campo *dirección destino* es indiferente y no se usa, lo que se indica con el tipo correspondiente; en la firma del bloque destino el campo *dirección origen* contiene los bits de referencia del salto, y en el tipo de firma se le indica al procesador de guardia que ha de verificar que el origen del salto es correcto.

En el tercer caso, para permitir que los bloques que contienen los saltos simples hagan la verificación utilizando la firma del bloque origen y, cuando el salto se produce desde el bloque con salto múltiple, se realice la verificación con la firma del bloque destino, se ha de llegar a un compromiso entre funcionalidad y cobertura de detección.

Cuando el salto tiene lugar desde el bloque con salto múltiple esta verificación sólo puede realizarse, igual que en el caso anterior, una vez alcanzado el bloque destino y con su firma. Esto fuerza a que el bloque destino indique, mediante los bits correspondientes del campo *tipo* que ha de verificarse el origen (que ha de corresponder al bloque con el salto múltiple).

Sin embargo, cuando el salto se produce desde uno de los bloques con salto sencillo, la verificación se hace utilizando la firma del bloque origen; el procesador de guardia encuentra, sin embargo, cuando obtiene la firma del bloque destino, que el tipo de ésta indica que se ha de realizar la verificación del origen del salto utlizando los bits *dirección origen*, lo cual es una incongruencia.

Llegados a este punto, se podían haber tomado dos caminos distintos en lo que a especificación del comportamiento del procesador de guardia se refiere, a saber: *a*) No contemplar este escenario como posible, y entender que, llegado el caso, la secuencia de firmas debería indicar un error en el flujo de ejecución, o *b*) incorporar este escenario como uno de los posibles, haciendo que el procesador de guardia acepte esta incongruencia en la secuencia de firmas como válida.

Se ha elegido la segunda opción, mayor funcionalidad o mayor número de casos tratables, a costa de reducir la posibilidad de detección de errores. Efectivamente, como se indica en la sección 2.3.2 el procesador de guardia

dispone de un mecanismo para detectar errores en el flujo de ejecución basado en la secuencia de tipos de firmas de referencia de los bloques que el procesador principal ejecuta; este mecanismo permite detectar saltos incorrectos teniendo en cuenta que no todas las combinaciones de tipos de firmas en una secuencia de ejecución son válidas. Pues bien, a pesar de que la secuencia de firmas para incorporar este caso al conjunto de escenarios tratados puede dar lugar a una de esas secuencias inválidas, se ha retirado este caso del mecanismo de detección de errores, pasando entonces a ser una secuencia válida más.

El cuarto caso, un bloque alcanzado desde dos o más instrucciones con salto múltiple, no puede ser verificado con ISIS. Para que la verificación pudiera tener lugar, sería necesario que el bloque destino tuviera no un campo *dirección origen*, sino uno por cada uno de los bloques con salto múltiple que hacen referencia a éste como uno de sus posibles destinos, lo que no es posible. Lo que se permite con ISIS es la existencia de este caso, a pesar de que no se pueda verificar el salto. Para ello, los bloques origen (con salto múltiple) no indican la verificación del destino en origen (pues, al ser un salto múltiple, exigiría un conjunto de campos *dirección destino*), y en el bloque destino no se indica la verificación del origen (que exigiría un conjunto de campos *dirección origen*).

Tampoco puede ser verificado el salto desde los bloques con salto múltiple del quinto y último escenario, solamente desde aquellos bloques con salto simple. Para éstos la verificación se hace con la firma del bloque origen y su campo *dirección destino*; para los primeros, el único tratamiento posible es el del caso anterior.

Parecería, con la descripción del tratamiento de los últimos casos, que la verificación de secuencias no válidas de las firmas de dos bloques consecutivos no es posible, pues el hecho de que un bloque origen indique o no la verificación con el campo *dirección destino* no implica un valor concreto de verificación en el destino.

Efectivamente esto es así, pero en el tipo de bloque se ha añadido, además de la indicación de la verificación o no del salto con el campo *dirección destino*, una indicación del tipo de bloque que el procesador de guardia debería encontrar en el destino. Este indicación fuerza a que el procesador de guardia

Cuadro 2.1: Casuística de saltos

Tipo de firma			
Bloque origen		Bloque destino	
Verificar destino	Verificar origen (en destino)	Verificar origen	Tratamiento
no	no	no	casos 4 ó 5; en este último, el bloque destino es alcanzado desde uno de los bloques con salto múltiple
no	no	sí	error
no	sí	no	error
no	sí	sí	caso 2 o caso 3, si el bloque destino es alcanzado desde el único bloque con salto múltiple
sí	no	no	casos 1 ó 5; en este último, el bloque destino es alcanzado desde uno de los bloques con salto sencillo
sí	no	sí	caso 3; el bloque destino es alcanzado desde uno de los bloques de salto sencillo, y el procesador de guardia, a pesar de lo indicado en la firma del bloque destino, no realiza la verificación de que el origen es correcto
sí	sí	sí	error
sí	sí	no	error

requiera (o no) de la firma de referencia del bloque destino, la verificación del origen del salto.

La casuística y el tratamiento que en cada caso da el mecanismo de detección de errores basado en la secuencia de firmas se indica en la tabla 2.1, en la que se detallan todos los casos posibles entre dos firmas consecutivas, a la izquierda la firma del bloque origen y en el centro la firma del bloque destino.

En la mencionada tabla, la primera columna indica si el salto desde el bloque origen es simple o múltiple: un salto simple se indica como "sí" en la primera columna, pues el destino del salto ha de comprobarse con la firma del bloque origen, y un salto múltiple como "no". En la segunda columna, la indicación del tipo de verificación del origen que se debe encontrar en la firma del bloque destino es la que permite detectar los errores de secuencia, si el bloque destino indica lo contrario.

Como se puede observar también en dicha tabla, para resolver la incongruencia del tercer escenario, el procesador de guardia no realiza la verificación del origen del salto cuando llega a un bloque destino si, utilizando la firma del bloque origen, se ha verificado que el salto es correcto, independientemente de lo indicado en la firma de referencia del bloque destino.

2.4. Soporte software

Uno de los aspectos más importantes que dotan de verosimilitud a una propuesta de estas características es el desarrollo del soporte software necesario para la inserción de las firmas de referencia.

Para este desarrollo práctico se ha elegido el compilador de libre distribución `gcc` en su versión 2.95 y el conjunto de programas y librerías que lo acompañan `binutils` en su versión 2.9.5.

La elección de este compilador como punto de partida no es casual. Por un lado, es uno de los pocos compiladores de código abierto; esta característica, junto con el hecho de ser desarrollado y mejorado por un buen número de voluntarios alrededor del globo, hace que sea un software relativamente bien documentado. Además, también es digno de mención el soporte que estas personas ofrecen, de forma completamente altruista, a aquellos que, como este autor, se asoman a la ingente cantidad de código fuente que lo conforma.

Por otro lado, la estructura interna con la que se ha desarrollado permite, de una forma relativamente sencilla, *portar* el compilador para dar soporte a nuevas arquitecturas, o a nuevos modelos de procesadores de arquitecturas existentes, permitiendo su integración de forma homogénea.

También hay que resaltar que, con el código fuente de este compilador, es sencillo disponer de un *compilador cruzado*: un compilador cuyos programas están preparados para ser ejecutados en una arquitectura *host*, pero que genera código binario ejecutable sobre una segunda arquitectura o *target*[1]. En particular, y entre la multitud de arquitecturas soportadas, se dispone de soporte para generar un compilador cruzado con la arquitectura MIPS como *target*. Esta arquitectura es la utilizada en el procesador sobre el que se realiza la parte práctica de este trabajo (véase el capítulo 3).

Por último, también es necesario recalcar que el compilador `gcc` no es una herramienta puramente académica: es un compilador que compite con muy buenos resultados por los puestos de cabeza en el ránking de los compiladores más utilizados en la industria, y muy especialmente en el ámbito de los sistemas empotrados.

En conjunto, se trata de una herramienta bien estructurada, relativamente bien documentada y que está pensada, justamente, para que la inclusión de nuevas arquitecturas o modelos de procesadores no sea una tarea imposible. Pero no se equivoque el lector: se trata de una herramienta que, en su conjunto, requiere una cantidad ingente de código fuente y que, para ser abarcada en su totalidad, necesitaría de un equipo humano considerable. Afortunadamente para el autor de este trabajo, la estructura de este compilador ha permitido que las modificaciones necesarias para la generación de las firmas de referencia estén bastante localizadas en unos pocos módulos.

2.4.1. Estructura interna del gcc

No es intención de este autor desmenuzar aquí, uno por uno, todos y cada uno de los elementos y programas que forman el compilador `gcc`. Sin embargo, sí resulta importante describir someramente los elementos más importan-

[1]Aunque no resulta relevante para este trabajo, cabe aquí mencionar una última posibilidad del compilador `gcc`, el cruce canadiense (*canadian cross*), que incluye el uso de tres arquitecturas diferentes: se trata de un compilador cuyo código fuente se genera (compila) en un *host*, que se ejecuta en un segundo *host*, y que genera código binario para una tercera arquitectura, el sistema final o *target*

tes que han sido modificados como resultado de este trabajo para conseguir la generación automática de las firmas de referencia para el procesador de guardia.

En primer lugar, cabe describir la estructura del código de este compilador como un conjunto de tres elementos:

- La librería estándar de soporte para el lenguaje C, `libc.a`, que contiene el código de las rutinas necesarias para todo programa que se ejecute en un entorno con sistema operativo, especialmente en lo que a entrada/salida se refiere. Para los sistemas empotrados, esta puede ser reemplazada por otra librería, denominada `newlib`, con un menor número de requisitos sobre el sistema operativo destino. Con esta última también se pueden desarrollar programas para sistemas *bare*, sin sistema operativo.

- El compilador propiamente dicho, que genera código ensamblador y que se apoya en la librería `libc.a`. En realidad se trata de un conjunto de programas, el primero de los cuales es el denominado preprocesador, `cpp`. Dentro del compilador `gcc` existe soporte para varios lenguajes de alto nivel, aunque para el trabajo que nos ocupa se ha limitado el uso al lenguaje C.

- El conjunto de utilidades conocido como `binutils`. Entre ellas podemos citar el programa ensamblador `gas`, el montador/cargador `ld`, el depurador/simulador `gdb`, etc. Todas estas utilidades descansan sobre una base común que da soporte al formato binario de los ficheros generados, tanto ficheros *objeto* resultado del ensamblado, como los ejecutables resultado del montador/cargador. Esta librería común se denomina `bfd`, y puede, en función de la arquitectura del sistema *target*, ofrecer soporte para los formatos *a.out*, *coff* (*Common Object File Format*) y *elf* (*Executable and Linkable Format*).

Lo que resulta de vital importancia para la instrumentación de los programas con las firmas de referencia es que el compilador traduce las sentencias de alto nivel (del lenguaje que sea) a ensamblador, generando toda la información para éste en formato de texto y utilizando etiquetas y nombres (símbolos), y que el proceso de ensamblado y montaje están separados de la compilación.

Es decir, que la instrumentación de los programas no requiere de la modificación del compilador, sólo de las herramientas que lo sustentan, las `binutils`.

Y que, cuando en el programa original se especifica que se ha de saltar a determinada instrucción, el compilador genera una instrucción de salto en ensamblador en la que el destino se especifica con una etiqueta, con un nombre (posiblemente generado de forma automática); aunque se modifique el conjunto de posiciones de memoria intermedias (añadiendo, por ejemplo, las firmas de referencia), basta con mantener el mismo nombre o etiqueta para que el salto se produzca a la instrucción correcta.

Es necesario recalcar en este momento que el uso de referencias simbólicas se mantiene durante todo el proceso, hasta el último momento si es necesario, gracias al uso de los *fix-up records*, descritos en la sección 2.4.2. Mantener estas referencias simbólicas es un requisito imprescindible para dar soporte a la creación de programas a partir de varios ficheros *objeto* y librerías, puesto que las direcciones de los símbolos externos no se conoce durante el proceso de ensamblado. No es, posiblemente, hasta el momento del montaje que se puede determinar dicha dirección (y se dice entonces que el símbolo se *resuelve*), momento en el que se puede sustituir la referencia simbólica por la numérica.

2.4.2. Elementos internos de las `binutils` más relevantes

Describir con detalle la estructura interna de los ficheros objeto, y la funcionalidad que ofrece el conjunto de herramientas conocido como `binutils` queda fuera del alcance de este trabajo.

Dicho esto, tampoco sería posible describir cómo se generan las firmas de forma automática ni el efecto que el proceso de compilación y ensamblado tienen sobre éstas sin describir, acaso someramente, algunos de los elementos internos. Estos elementos no son utilizados en exclusiva para la generación de las firmas, ni se han creado con tal fin. Ya estaban incorporados a las `binutils` por otras razones; lo que se ha hecho ha sido aprovecharlas en un caso, o tenerlas en cuenta, en el otro, para que la generación de las firmas sea posible.

Fix-up records

Un *fix-up record* es un elemento incorporado al formato binario interno utilizado por las herramientas `binutils` para mantener una referencia simbólica. Está asociado a una instrucción en particualr, y no se trata solamente de mantener la información del símbolo al que se quiere hacer referencia, sino

mantener también un indicador, un índice para indexar una tabla de procedimientos de cálculo.

Hay que tener en cuenta que, para cuando el símbolo referenciado puede ser resuelto y sustituido por su dirección, en muchos casos no se trata de incorporar al programa esta dirección sin más; es necesario que esta dirección sea procesada, manipulada e insertada como un campo más de una instrucción en particular, y que dicha inserción no debe modificar los demás campos de la misma.

Por ejemplo, en la arquitectura MIPS, en la que todas las instrucciones son de 32 bits, en los saltos condicionales se especifica el destino del salto como el desplazamiento (con signo y contabilizado en instrucciones de 32 bits) entre la instrucción de salto y el destino; este desplazamiento se incorpora en la parte baja de la instrucción de salto como un campo de 16 bits. Así pues, una instrucción de salto condicional que hace una referencia simbólica al destino del salto contiene, desde el momento de su generación, la mitad (superior) de sus 32 bits ya codificados de forma permanente.

Cuando el símbolo referenciado puede ser resuelto, el montador necesita determinar cómo la dirección de dicho símbolo encaja en la instrucción a la que el *fix-up record* hace referencia. En el caso de la instrucción del ejemplo, un índice numérico permite al montador indexar una tabla de procedimientos de cálculo y modificación de instrucciones, y será este procedimiento el que realice el cálculo del desplazamiento y modifique (la parte baja de) la instrucción en cuestión.

El mantener separado, por un lado la resolución de los símbolos en direcciones y por otro los procedimientos para manipular dichas direcciones e incorporarlas en las instrucciones permite, de manera muy sencilla, incorporar nuevos tipos de cálculos: Basta con crear una nueva entrada en la tabla de procedimientos de cálculo y asignarle un índice.

Desde ese momento, cualquier herramienta de las `binutils` puede insertar o resolver este nuevo tipo de *fix-up record*. Precisamente lo que la generación de firmas requiere es un nuevo procedimiento de cálculo para los saltos: la generación de los bits en los campos *dirección origen* y *dirección desino* en las firmas de referencia.

Es evidente que, para cada arquitectura soportada por las `binutils`, existe un conjunto diferente de *fix-up records* disponibles. Por tanto, para que la técnica ISIS pueda ser transportada a otra arquitectura habría que replicar

los *fix-up records* creados para la generación de las firmas. Sólo podrían reutilizarse los ya creados si el procedimiento de cálculo se mantiene (lo cual sí se cumple), y los campos de dirección se mantienen con la misma longitud y en la misma posición dentro de las firmas de referencia.

Variant frags

Un *variant frag* es un fragmento de código de longitud variable. La variabilidad viene impuesta por la necesidad de generar un código óptimo unida al desconocimiento de cuál es la secuencia óptima para un caso particular en el preciso momento de la generación de dicho código.

Se podría optar por la vía más conservadora, generando aquella secuencia que siempre es posible utilizar, aún a costa de resultar ineficiente en la mayoría de los casos, en los que la tarea podría resolverse con menos instrucciones.

Lo que se pretende con un *variant frag* es retrasar la elección de cuál es la secuencia óptima de instrucciones hasta tener toda la información necesaria para tomar la decisión de forma razonada. Dicho así, parecería que parte del proceso de ensamblado (la generación de algunas instrucciones) debe retrasarse hasta disponer de la información necesaria, quizá hasta el momento del montaje. Pero esto es incoherente con respecto a una estructura por capas en la que cada herramienta se encarga de una parte del trabajo.

¿Cómo se solventa esta disyuntiva? Generando las diferentes alternativas posibles de secuencias de instrucciones, indicando para cada una de ellas qué requisitos necesita y cuántas instrucciones implica. De esta forma, la generación de instrucciones se circunscribe al ensamblador, pero la elección de cuál es la secuencia óptima puede retrasarse. Esto es posible porque, generalmente, la secuencia que se considera óptima es la que necesita menos instrucciones.

Pongamos un ejemplo, utilizando esta vez una instrucción de lectura de memoria de la arquitectura MIPS. Para referenciar una posición de memoria en esta arquitectura se usa lo que se conoce como direccionamiento indexado: un desplazamiento (con signo y de un tamaño máximo de 16 bits) a partir de una dirección denominada base (almacenada en un registro base). Tanto el desplazamiento como el registro base se explicitan en la instrucción de acceso a memoria. El problema para el programa ensamblador es saber, en el momento de generar la secuencia de instrucciones de acceso a memoria, si existe un registro base disponible y cargado con una dirección base

tal que, mediante un desplazamiento de 16 bits, se pueda acceder al dato correspondiente.

Si se trata de acceder a una variable local de la función que se está compilando, el registro base existe, sea el puntero de pila o el *frame pointer*, y el programa ensamblador dispone de toda la información necesaria para generar, habitualmente, una única instrucción (la de lectura de memoria) para acceder al dato. Caso de que el desplazamiento no se pueda codificar con 16 bits, el ensamblador genera una secuencia diferente, alternativa, en la que se carga un registro con la dirección del dato en memoria y luego ese registro se usa como registro base en la instrucción de lectura de memoria (con desplazamiento cero). Evidentemente, esta última secuencia requiere de más instrucciones y el programa ensamblador, por razones de eficiencia, intenta evitarla en la medida de lo posible.

Sin embargo, si el dato que se quiere cargar de memoria está referenciado de forma simbólica y con un nombre que el ensamblador no puede resolver en ese momento, ¿qué se puede hacer? El programa ensamblador crea un *variant frag* con las dos secuencias de instrucciones, cada una con los *fix-up records* necesarios para que pueda ser completada.

Cuando el símbolo puede ser resuelto, se procede a realizar los cálculos de cada *fix-up record*, comenzando por aquella secuencia de menor longitud. Si el cálculo no falla, se descarta la secuencia mayor. Si el cálculo falla, porque el desplazamiento no puede ser codificado en el campo asociado de la instrucción de lectura de memoria, la secuencia menor es descartada y se usa la mayor.

De esta manera, se consigue que la generación de instrucciones sea competencia exclusiva del programa ensamblador y que la secuencia de instrucciones utilizada para resolver cada tarea sea siempre la óptima, aún a pesar de que el ensamblador no disponga de toda la información que la elección correcta requiere.

Los *variant frag* no ayudan a la generación de las firmas; antes bien, su uso dificulta el trabajo de la generación de las mismas.

En efecto, que un bloque de instrucciones incluya un *variant frag* implica, por ejemplo, que no se pueda saber a ciencia cierta cuál será el número de instrucciones del mismo. Al mismo tiempo, un bloque de muchas instrucciones debe ser partido en varios de menor longitud para acomodarlos a lo que se permite en el campo *longitud* de las firmas de referencia.

En este caso se ha optado por la solución más sencilla, aunque menos eficiente:

de cada *variant frag* se ha tomado en consideración la secuencia más larga para determinar cuándo es necesario subdividir un bloque. El efecto negativo es que, cuando finalmente se escoge la secuencia de menor longitud del *variant frag*, la longitud total del bloque puede ser menor de la máxima; y si esto se hubiese podido tener en cuenta quizá no habría sido necesario dividir el bloque en dos, una división que genera una sobrecarga tanto en prestaciones como en consumo de memoria.

2.4.3. Inserción automática de las firmas de referencia

Como se ha mencionado anteriormente, este desarrollo práctico utiliza la versión 2.95 del compilador `gcc` y modifica las herramientas que lo acompañan, las `binutils`, en su versión 2.9.5. Se ha tomado como punto de partida la versión de este software como compilador cruzado, utilizando la arquitectura i386 sobre el sistema operativo linux como *host* y la arquitectura MIPS sobre un sistema *bare* o sin sistema operativo como *target*.

Puesto que este desarrollo es, sencillamente, una prueba de que la generación automática de las firmas requeridas por la técnica ISIS es viable, se ha restringido esta generación al formato binario *elf* (acrónimo de *Executable and Linkable Format*) en su versión *big-endian*.

La inserción de las firmas de referencia no resulta compleja gracias al uso de los *fix-up records*. De hecho, cuando se inserta una firma precediendo a un bloque de instrucciones, prácticamente todos los campos quedan vacíos a la espera de que la firma sea "parcheada" con los *fix-up records* necesarios.

En efecto, en muchos casos no se puede establecer de forma definitiva la longitud del bloque, debido al uso de *variant frags*. Los bits de los campos *dirección origen* y *dirección destino* necesitan resolver los símbolos correspondientes, y en cuanto a la firma de respaldo de los patrones binarios de las instrucciones, es obvio que no puede ser calculada hasta que todos los bits de todas las instrucciones han sido determinados; esto incluye la resolución de todos los *fix-up records* y *variant frags* que pudiera haber.

Esta dependencia ha forzado a crear un montador en dos pasadas. La primera pasada resuelve todos los símbolos del programa, excepto los referidos a las firmas de referencia; hay un gran esfuerzo en el desarrollo del montador para que éste pueda resolver su tarea en una única pasada sobre el código. En la segunda pasada, y ya con todas las instrucciones en su forma definitiva, es

cuando se pueden completar todos los campos de las firmas de referencia.

Para los *fix-ups* correspondientes a la dirección destino de los saltos, tanto condicionales como incondicionales, se utiliza el mismo símbolo referenciado en la propia instrucción de salto. En el caso de los saltos con múltiples destinos, es necesario que en el momento de la inserción de la instrucción el programa ensamblador tenga la información de la especial característica del salto, y de todos los posibles destinos.

A fin de no tener que modificar la parte de compilación de alto nivel, que habría de generar dicha información bien de forma automática bien con el apoyo del programador, los únicos saltos de este tipo que se han incorporado al soporte software son los retornos de procedimiento. Estos saltos múltiples no requieren de ninguna información específica desde el compilador, y es por esto que se han incluido en el soporte software.

Durante el ensamblado, cuando se insertan en el código objeto las últimas instrucciones de un procedimiento, el programa ensamblador modificado genera de forma automática y específica para esta tarea un símbolo que incluye el nombre del procedimiento. Este símbolo es utilizado como referencia para la codificación del campo *dirección origen* en la firma de referencia del bloque siguiente al de la instrucción de llamada a procedimiento (el que se ejectua tras el retorno).

Para que el final del procedimiento pueda ser utilizado como dirección de referencia es necesario que, para cada procedimiento o función, exista un único punto de salida. Este es un requisito imprescindible, pues en la técnica ISIS se da por supuesto que, una vez se ejecuta un procedimiento, existe una única instrucción con la que se retorna del mismo.

Este requisito no es nuevo, y de hecho no ha sido necesario forzarlo con la modificación que aquí se describe, pues ya estaba incorporada en el código del ensamblador para la generación del *epílogo* de la función. Este epílogo incluye el código para restaurar los registros modificados por la función, recuperar de la pila el espacio de las variables locales, etc. Por simple economía, es necesario que dicho epílogo no se repita aunque en el texto fuente de alto nivel una función pueda retornar desde múltiples puntos del código; lo que hace el compilador es redireccionar esos retornos hacia un epílogo común desde el que efectivamente se realiza el retorno.

Sin embargo, si se realizara la programación directamente en ensamblador, el programador debe ser consciente de este requisito. De lo contrario, el procesa-

dor de guardia detectará como errores retornos correctamente ejecutados por el procesador principal, pero realizados desde puntos de la función distinto de las últimas instrucciones.

Quizá queda por resaltar que, de hecho, los campos de *longitud* y *firma derivada* no requieren de ninguna clase de *fix-up*, pues basta con explorar y procesar el código hasta encontrar la siguiente firma de referencia para determinar su contenido.

2.4.4. Uso práctico del compilador

A partir del código fuente de las binutils modificadas, es necesario crear dos juegos de librerías de soporte. Uno de ellos sin firmas ni instrucciones insertadas, que se usará en el caso de desear la generación de código estándar de la arquitectura MIPS.

El segundo juego de las librerías de soporte incluye las firmas de referencia de los bloques, además de los *fix-ups records* necesarios para el parcheado de las firmas en el momento de generar el código ejecutable.

Es contra este último juego de librerías que hay que compilar los programas de usuario si se desea que el programa, en su conjunto, incorpore las firmas de referencia de la técnica ISIS.

La elección de qué librerias se utilizan, y de si se han de insertar o no las firmas de referencia se explicita a través de opciones (*switches*) en la línea de órdenes en el momento de la compilación/montaje.

La posibilidad de compilar código sin incluir las firmas de referencia permite utilizar este compilador para generar código estándar de la arquitectura MIPS; esto puede ser aprovechado para utilizar el compilador de forma que la incorporación de las firmas en el código de un sistema se realice de forma progresiva. Como se verá en el capítulo 3, el procesador que se ha implementado para hacer realidad la propuesta ISIS permite que, en tiempo de ejecución, se indique si la tarea en ejecución lleva o no las firmas de referencia, y, por ende, si el procesador de guardia puede verificar su ejecución o no.

En caso de utilizar la compilación con la inserción de las firmas de referencia, durante el proceso se generan mensajes de información que indican, por ejemplo, cuántos bloques tenía el programa originalmente, cuántos han sido troceados, cuántas instrucciones y firmas han sido insertadas en el pro-

grama, etc. Esta información será crucial para analizar la sobrecarga en los requerimientos de memoria que el uso de ISIS impone.

2.5. Conclusiones

En este capítulo se han analizado los puntos débiles de las propuestas existentes en materia de procesadores de guardia. De las conclusiones de dicho análisis surge la propuesta de una nueva técnica de empotrado de firmas de referencia, a la que se ha denominado ISIS (acrónimo de *Interleaved Signature Instruction Stream*). Con esta propuesta no es necesario modificar el juego de instrucciones del procesador principal, ni se incurre en graves penalizaciones en las prestaciones finales, debido a que el procesador principal no hace referencia, ni ejecuta, dichas firmas.

Para conseguir aislar las firmas de referencia de las instrucciones del procesador principal, manteniendo estas firmas intercaladas entre dicho código, es necesario realizar algunas modificaciones sencillas a la semántica, que no a la codificación, de algunas instrucciones. Específicamente, se ven afectadas por estos cambios semánticos las instrucciones de salto condicional y las de salto a subrutina.

Este aislamiento entre el procesador principal y las firmas de referencia persigue reducir la pérdida de prestaciones en la que incurren otras propuestas previas de procesadores de guardia. Sin embargo, a veces es necesario interferir en el programa original para insertar instrucciones de salto. Estas instrucciones no suponen sólo un incremento en los requisistos de memoria, también afectan a las prestaciones del sistema resultante.

La técnica ISIS no requiere de la arquitectura que se pretende monitorizar ningún cambio en el juego de instrucciones ni ninguna característica especial, lo que permite su aplicación a cualquier arquitectura.

A partir de la técnica de empotrado de firmas se desarrolla una firma de referencia que incorpora un conjunto de mecanismos de detección de errores; estos mecanismos están basados, fundamentalmente, en la utilización de bits de respaldo tanto para las instrucciones que componen el bloque como para los saltos entre bloques.

La implementación de uno de los mecanismos de detección de errores descritos, el denominado *inicio de bloque* no es siempre posible. Para que su

uso sea factible es necesario que la codificación de las firmas de referencia esté completamente diferenciada de las instrucciones del procesador principal; como uno de los objetivos de ISIS es que sea aplicable al mayor número de arquitecturas posibles, esta codificación, que no siempre es posible pues depende de los códigos de operación dentro del juego de instrucciones que no estén en uso, no es un requisito impuesto sino una recomendación. Si dicha codificación no es posible la técnica de empotrado de firmas ISIS puede igualmente emplearse, aunque en las firmas de referencia se perderá uno de los mecanismos de detección de los errores de los saltos del procesador.

De entre estos mecanismos, destaca por su originalidad la verificación de un salto con múltiples destinos tras el salto, utilizando los bits de referencia del bloque destino. Esta es una solución, como se ha dicho, original, al problema de la verificación de saltos con destino múltiple, que permite verificar escenarios de salto que no habían sido contemplados previamente.

Aún quedan algunos casos o escenarios de salto que no pueden ser verificados: el caso de un bloque de instrucciones alcanzado desde varias instrucciones de salto con destino múltiple. Dicho con otras palabras, un bloque que es un destino compartido por varios de estos saltos.

Por último, y como demostración práctica de la viabilidad de la propuesta anterior, se ha descrito cómo partiendo de un compilador de código abierto y utilizado ampliamente tanto en el entorno académico como el industrial, y mediante una serie de modificaciones, se generan de forma automática las firmas de referencia.

Con esta implementación del software de soporte se pueden compilar programas escritos en lenguaje C, que posteriormente son instrumentados de forma automática durante el ensamblado; en general, la información necesaria para completar la codificación de las firmas de referencia no está disponible hasta el momento de generar el programa ejecutable, momento en el que las firmas son definitivamente codificadas.

Aunque el soporte software permite contemplar todos los escenarios de salto propuestos, y para evitar la modificación de la parte de compilación en alto nivel, los únicos saltos con múltiples destinos a los que se ha dado soporte es a los retornos de procedimiento. Cualquier otro salto múltiple requeriría que, bien el usuario, bien el compilador de alto nivel, informaran al programa ensamblador de qué saltos tienen esta característica, y cuáles son los destinos posibles.

Lo que queda por hacer es, en primer lugar, demostrar de forma práctica que las alteraciones semánticas propuestas para las instrucciones son factibles, y que un sistema así modificado es viable. En segundo lugar, demostrar que los mecanismos de detección de errores incorporados son eficaces. Y, por último, determinar la magnitud de la sobrecarga, tanto en consumo de memoria como en prestaciones, en la que se incurre por utilizar esta técnica.

El capítulo 3 se dedica al primero de estos trabajos: describir el desarrollo de un procesador, un modelo de la arquitectura MIPS desarrollado utilizando el lenguaje de descripción de hardware VHDL, dotado de un procesador de guardia y modificado para utilizar la técnica ISIS.

Capítulo 3

HORUS: Implementación de la técnica ISIS

To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users.

D. P. Siewiorek [7]

3.1. Introducción

En el capítulo 2 se ha propuesto una técnica para empotrar las firmas de referencia de un procesador de guardia, intercalándolas entre los bloques de instrucciones del procesador principal.

Para minimizar la pérdida de prestaciones en la que se incurre por la monitorización, la técnica ISIS propone aislar completamente las secuencias de instrucciones del procesador principal y las firmas de referencia. Para conseguir este aislamiento, es necesario realizar una serie de modificaciones al comportamiento del procesador en lo que al tratamiento de saltos condicionales y llamadas a procedimiento se refiere.

En este capítulo se presenta una implementación de dicha propuesta sobre un procesador RISC, utilizando como base la arquitectura MIPS, para la que

se dispone de soporte software para el cálculo y la generación automática de las firmas de referencia.

Este desarrollo persigue, como principal objetivo, demostrar que las modificaciones propuestas son viables. También ha de servir como banco de trabajo para obtener índices de prestaciones sobre programas de ejemplo.

A este procesador se le ha dado el nombre HORUS[1]. Se trata de un modelo desarrollado utilizando el lenguaje de descripción de hardware VHDL, utilizando un subconjunto de elementos del lenguaje que permite que el resultado sea sintetizable, listo para ser descargado sobre un dispositivo lógico programable tipo FPGA.

3.2. Banco de pruebas

Para poder realizar las pruebas necesarias que permitan la verificación del funcionamiento de HORUS es necesario construir un banco de pruebas o *testbench* que debe incluir, al menos, el propio sistema a verificar junto con los elementos necesarios que soportan la ejecución de programas de éste. A saber: la memoria donde residirán tanto las instrucciones del programa en ejecución como los datos que se manejan, y el sistema de reloj que indica el paso del tiempo para todo el sistema.

Los modelos desarrollados para estos elementos no son modelos sintetizables, ni es necesario que lo sean. Se trata de dotar al modelo de HORUS de los elementos de soporte necesarios para que pueda ejecutar un programa, y de esta forma verificar que su funcionamiento es correcto.

En particular, el modelo desarrollado en VHDL para la memoria del sistema permite indicar, en el momento de la simulación, el nombre de un fichero de texto que contiene, en un formato estándar, el contenido inicial de la memoria antes de que el procesador inicie la ejecución del programa. De esta manera dicha memoria permite modelar una memoria de tipo ROM con las instrucciones del programa a ejecutar, cuyo contenido ha sido programado de forma previa a la ejecución como ocurre de típicamente en los sistemas empotrados.

[1]Puesto que este procesador deriva de la técnica ISIS (podríamos decir que es fruto de la misma), nombre que coincide con el nombre de una antigua divinidad egipcia, el nombre del procesador pretende seguir la genealogía de la antigua mitología egipcia, en la que HORUS es hijo de los dioses ISIS y OSIRIS.

El nombre del fichero se indica mediante lo que en el lenguaje VHDL se conoce como *genérico*. Un génerico en VHDL es una constante utilizada durante la compilación y/o simulación del modelo en VHDL y que permite modificar la estructura interna o la lógica del propio modelo en función del valor indicado por dicha constante. Un ejemplo prototípico sería la especificación, mediante un genérico, del número de bits de un registro; el uso de un genérico permite crear un registro parametrizable, en el que se establece el número de bits efectivo cuando el registro se incorpora en un modelo de mayor envergardura (se dice, entonces, que es *instanciado*).

Al genérico que permite dar nombre al fichero con el contenido inicial de la memoria se le ha llamado `filename`. El formato elegido para representar dicho contenido es el formato estándar intel-hexadecimal, utilizado habitualmente para la representación de contenidos de memorias (especialmente cuando se trabaja con programadores de pastillas de memoria ROM). Se trata de un formato orientado al procesamiento por líneas y en el que direcciones y contenidos están representados en hexadecimal por texto ASCII legible, lo que permite que el desarrollo de un traductor en VHDL (o en cualquier lenguaje de programación de alto nivel) no resulte demasiado farragoso, pues además de manejar texto el número de registros distintos (tipos distintos de líneas de texto) a tratar es escaso.

La elección de este formato se asienta, además de su simplicidad, en que está entre los formatos soportados por las herramientas `binutils` del compilador `gcc` que permiten la transformación de un fichero objeto de uno a otro formato.

Así, tras el montaje del programa, en el que se obtiene un fichero binario con el formato por defecto (elf), basta con invocar a la utilidad `objcopy` para transformar el ejecutable al formato intel-hexadecimal. De esta forma, se puede también, mediante la simulación de la ejecución del programa, verificar que la inserción de firmas se ha realizado correctamente, validando así las modificaciones al compilador estándar.

También con el objeto de facilitar la verificación del modelo se ha incluido un sistema de traza, directamente conectado al pipeline del procesador, que genera la información más relevante de la instrucción que ocupa cada etapa. Puesto que este elemento depurador no es sintetizable pero se encuentra directamente conectado con el procesador principal, su posición en la organización jerárquica del modelo es bastante bastante peculiar, ya que al no ser sintetizable debería quedar fuera de los límites del modelo que representa al dispositivo lógico programable (FPGA o similar).

Para solventar este aspecto en el modelo del sistema se ha utilizado otro genérico, al que se ha dado el nombre `debug`, para que el sistema de traza esté presente o no en el modelo. Es interesante que esté presente cuando se simula el sistema, pues ayuda a verificar su funcionamiento, pero evidentemente no puede estar presente cuando se pretende sintetizar el modelo para descargarlo sobre un chip real, pues el código no es sintetizable.

Por último, se ha utilizado un tercer genérico llamado `isis` que permite eliminar del sistema todo rastro del procesador de guardia: el procesador de guardia en sí, el segundo puerto de lectura de la memoria caché de instrucciones, las conexiones con el procesador principal, los registros asociados del coprocesador de control del sistema, etc.

Con este último genérico se van a poder realizar de forma sencilla los análisis de coste, en términos de área de silicio, de esta implementación al utilizar el procesador de guardia.

Es importante resaltar que el efecto de este genérico no tiene nada que ver con la modificación del contenido del registro de sistema que permite, en tiempo de ejecución, activar o desactivar la verificación de firmas de cada una de las tareas del sistema. Mientras que este último permite una elección realizada por el software del propio sistema, durante la ejecución de un programa, y que requiere la presencia del procesador de guardia para ser efectivo, el genérico lo que hace es eliminar todo rastro del procesador de guardia del sistema, siendo entonces imposible su activación/desactivación por medio de software.

En definitiva, la declaración del testbench en VHDL del sistema, entidad a la que se ha dado el nombre `Test_SystemOnChip` queda, con los genéricos que se han comentado anteriormente, como

```
ENTITY Test_SystemOnChip IS
  GENERIC (
    filename: STRING  := "./memory.hex";
    debug:    BOOLEAN := FALSE;
    isis:     BOOLEAN := FALSE );
END Test_SystemOnChip;
```

En esta declaración los valores asociados a los genéricos son sus valores por defecto. Estos son los valores que se utilizan cuando se sintetiza el sistema; cuando se realiza una simulación, el valor de cada parámetro puede ser alterado sin necesidad de recompilar el código fuente; caso de no especificar en

la línea de órdenes un valor concreto para un genérico, se utiliza el valor por defecto.

Así, por ejemplo, para invocar la simulación del testbench desde la consola del simulador comercial modelsim, se puede hacer con la siguiente orden (el orden de los genéricos en la línea de órdenes no es importante)

```
vsim -Gfilename="./test/programas/memoria.hex"
    -Gdebug=true -Gisis=true Test_SystemOnChip
```

donde `vsim` es la orden que invoca al simulador, `-G` indica que lo que sigue a continuación es la modificación de un genérico de la entidad a simular utilizando la sintaxis *generico=valor* y `Test_SystemOnChip` es la entidad a simular; en este caso particular se ha modificado el valor de todos los genéricos del testbench respecto de su valor por defecto.

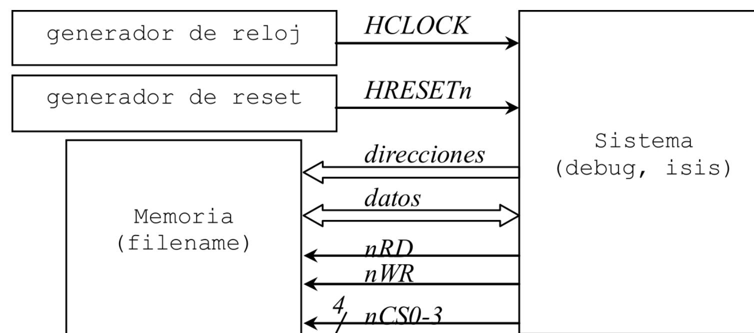

Figura 3.1: Organización del banco de pruebas

3.2.1. Organización del banco de pruebas

En la figura 3.1 se muestra la organización interna del banco de pruebas y los componentes que lo forman. En el mismo se puede observar también la influencia de cada genérico sobre los diferentes componentes.

Tanto el bus de direcciones como el de datos son de 32 bits. En el acceso a los datos de la memoria del sistema, cada *bytelane* puede ser activado individualmente, lo que permite realizar escrituras de datos con un tamaño inferior a 32 bits.

Como se puede observar por la interfaz de la memoria, ésta se ha modelado como una memoria estática de tipo RAM estándar. En este tipo de memorias,

y puesto que no existe una línea de reloj que sincronice las transferencias, la única garantía de que las operaciones se llevan a cabo correctamente es verificando, mediante el análisis temporal correspondiente, que el sistema cumple todos los requisitos temporales exigidos por la memoria: ancho de pulso de las señales de control, tiempos de establecimiento y mantenimiento de los datos, etc.

Puesto que de la memoria se ha realizado un modelo funcional y no con el nivel de detalle necesario como para incluir estos tiempos, en un sistema real la frecuencia del sistema debería ser lo suficientemente baja como para que las operaciones de lectura y escritura puedan realizarse en un ciclo de reloj.

Evidentemente, si la memoria del sistema real no fuera del tipo RAM estática, dentro del sistema habría que modificar la parte correspondiente a la interfaz con la memoria externa. Puesto que dicho interfaz se conecta a un bus interno estandarizado (denominado AMBA, véase la sección 3.3), el resto del sistema podría mantenerse intacto.

3.3. Arquitectura del sistema

El sistema (el *System-On-a-Chip*) diseñado tiene como núcleo principal al procesador HORUS, un procesador RISC con un pipeline de diseño clásico al que se ha añadido un procesador de guardia para verificar la integridad de los programas ejecutados.

Además del procesador de guardia, y como todo procesador de la arquitectura MIPS, se ha incorporado un coprocesador para el control del sistema (el denominado *System Control Coprocessor* o *Coprocesador 0* en la nomenclatura MIPS). En éste se encuentran los registros de control de la unidad de gestión de memoria (MMU, *Memory Management Unit*) y la gestión de las interrupciones y excepciones, entre otros registros; en algunos modelos de la arquitectura MIPS se ubican en este coprocesador los registros asociados al sistema de depuración, de índices de prestaciones, control de las memorias caché, etc. En el caso de HORUS, el coprocesador 0 permite el acceso a los registros del coprocesador de guardia para el correcto funcionamiento del sistema durante el procesamiento de las excepciones.

A este bloque se le ha añadido una memoria caché de instrucciones con dos puertos de lectura. Este acceso dual es necesario para permitir que el procesador de guardia obtenga las firmas de referencia de los bloques ejecutados por

el procesador principal sin perturbar la búsqueda de instrucciones de éste.

Para mantener la independencia del diseño de la caché de instrucciones respecto de los diferentes tipos de memorias con los que se podría dotar al sistema, la conexión a las mismas se realiza a través de un bus estándar denominado AMBA. El bus AMBA es un bus de altas prestaciones diseñado por la compañía ARM para la interconexión de diferentes elementos dentro de una pastilla; dicho bus se contempla la posibilidad de que existan varios maestros (o iniciadores de transferencias) conectados mediante el bus a uno o más esclavos. En la especificación de AMBA se distinguen dos segmentos: el segmento de altas prestaciones AHB (AMBA High-speed Bus) y el segmento de periféricos, de menor velocidad, denominado APB (AMBA Peripheral Bus). Ambos segmentos se conectan entre sí mediante un puente AHB-APB que permite a los maestros conectados al segmento AHB hacer uso de los periféricos del segmento APB.

La caché de instrucciones dispone de un controlador que, en caso de fallo por parte del procesador principal o el de guardia, rellena los contenidos de la memoria caché accediendo a la memoria del sistema a través del bus AMBA. Es, por tanto, un controlador de memoria caché y un maestro en el bus AMBA, lo que le permite ser el iniciador de las transferencias con memoria. En caso de que existan peticiones simultáneas de rellenado de la caché por parte de los dos procesadores, el acceso es serializado por este controlador, siendo siempre promocionado el procesador principal.

El acceso de HORUS a la memoria de datos se realiza a través de un sencillo maestro de bus AMBA, sin que exista en esta caso memoria caché alguna. No se ha incluido una memoria caché de datos por dos motivos: por un lado no resulta relevante para demostrar la viabilidad del trabajo que se propone en esta tesis, pues en nada se ven afectados los datos ni su tratamiento con la incorporación de un procesador de guardia; por otro lado, al no disponer de memoria caché de datos se incrementa el tráfico con la memoria externa (y, por tanto, a través del bus AMBA), lo que permite estresar fácilmente el sistema para determinar mejor el impacto que sobre las prestaciones tiene el uso del procesador de guuardia.

Para la gestión del bus AMBA se ha diseñado un árbitro de bus siguiendo las especificaciones de la compañía ARM. Este árbitro serializa el acceso de los diferentes maestros al bus, y un segundo elemento central denominado controlador de esclavos permite seleccionar a un esclavo de entre los conectados al bus como el destinatario de la transferencia de memoria. Este controlador determina el esclavo implicado en la transferencia en función de la direc-

ción de memoria utilizada por el maestro, según una tabla fijada para cada sistema en particular.

En el caso que nos ocupa, el único esclavo AMBA diseñado es el controlador de memoria que da acceso a la memoria externa.

Además de estos elementos también se ha diseñado un puente AMBA AHB-APB, entre el segmento de altas prestaciones AHB y el de menor capacidad de transferencia APB lo que permitirá, en un futuro, dotar al sistema de periféricos típicos de un sistema empotrado (comunicaciones, temporizadores, etc) bien mediante el uso de diseños propios bien mediante la implantación de periféricos diseñados por terceros para su conexión a un bus APB estándar.

Tanto el árbitro de bus como el controlador de esclavos se ha diseñado pensando en un sistema de mayor envergadura, con más de dos maestros y un esclavo. Esto permite de forma sencilla y directa, por ejemplo, que varios procesadores HORUS compartan el bus AMBA y los esclavos que a este bus se conecten, creando un sistema multiprocesador.

Parte II:

Publicaciones

A Watchdog Processor Architecture with Minimal Performance Overhead

Francisco Rodríguez, José Carlos Campelo, and Juan José Serrano

Grupo de Sistemas Tolerantes a Fallos - Fault Tolerant Systems Group,
Departamento de Informática de Sistemas y Computadoras,
Universidad Politécnica de Valencia, 46022-Valencia (Spain),
{prodrig, jcampelo, jserrano}@disca.upv.es,
WWW home page: http://www.disca.upv.es/gstf

Abstract

Control flow monitoring using a watchdog processor is a well-known technique to increase the dependability of a microprocessor system. Most approaches embed reference signatures for the watchdog processor into the processor instruction stream creating noticeable memory and performance overheads. A novel watchdog processor architecture using embedded signatures is presented that minimizes the memory overhead and nullifies performance penalty on the main processor without sacrificing error detection coverage or latency. This scheme is called *Interleaved Signature Instruction Stream* (ISIS) in order to reflect the fact that signatures and main instructions are two independent streams that co-exist in the system.

1. Introduction

In the "Model for the Future" foreseen by Avizienis in [1] the urgent need to incorporate dependability to every day computing is clear: "Yet, it is alarming to observe that the explosive growth of complexity, speed, and performance of single-chip processors has not been paralleled by the inclusion of more on-chip error detection and recovery features".

Efficient error detection is of fundamental importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent *Error Detection Mechanism* (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experiments demonstrate [2, 3, 4, 5], a high percentage of non-overwritten errors results in control flow errors.

Siewiorek states in [6] that "To succeed in the commodity market, faul-tolerant techniques need to be sought which will be transparent to end users". A fault-tolerant technique can be considered transparent only if results in minimal performance overhead in silicon, memory size or processor speed.

Although redundant systems can achieve the best degree of fault-tolerance, the high overheads implied limit their applicability in every day computing elements.

The work presented here provides concurrent detection of control flow errors with no performance penalty and minimal memory and silicon sizes. No modifications are needed in the instruction set of the processor used as testbed and the architectural ones are so small that they can be enabled and disabled under software control to allow binary compatibility with existing software. The watchdog processor is very simple, and its design can be applied to other processors as well.

The paper is structured as follows: The next section is devoted to present a set of basic definitions and it is followed by the outline of related works in the literature. Section 4 presents the system architecture where the watchdog is embedded. Section 5 discusses error detection capabilities, signature characteristics and placement, and modifications needed into the original architecture of the processor. A memory overhead comparison with similar work is performed afterwards, to finish with the conclusions.

2. Basic Definitions

The following definitions are taken from [5]:

1. A **branch** instruction is an instruction that can break the sequential flow of execution like a procedure call, a conditional jump or a return-from-procedure instruction.
2. A **branch-in** point is an instruction used as the destination of a branch instruction or the entry point of, for example, an interrupt handler.
3. A program is partitioned into branch-free intervals and branch instructions. The beginning of a branch-free interval is a branch-in instruction or the instruction following a branch. A branch-free interval is ended by a branch or a branch-in instruction.
4. A **basic block** is only a branch-free interval if it is ended by a branch-in. It is the branch-free interval and its following branch instruction otherwise.

With the definitions above a program can be represented by a *Control Flow Graph* (CFG). Vertices in this graph are used to represent basic blocks and directed arcs are used to represent legal paths between blocks. Figure 1 shows some examples for simple High Level Language constructs.

We call **block fall-through** to the situation where two basic blocks are separated with no branch-out instruction in between. Blocks are divided only because the first

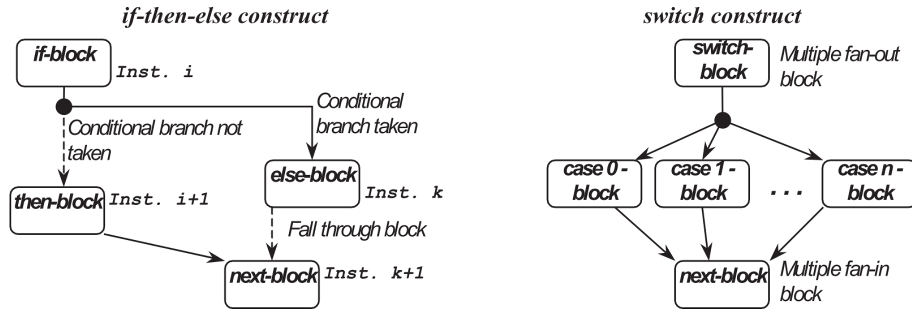

Figure 1: CFGs for some HLL constructs

branch-free interval is ended by a following branch-in instruction that starts the second block.

In [7] a block that receives more than two transfers of control flow it is said to be a *branch fan-in* block. We distinguish wether the control flow transfer is due to a non-taken conditional branch (that is, both blocks are contiguos in memory) and say that a **multiple fan-in** block is reachable from more than one out-of-sequence vertex in the CFG.

A branch instruction with more than one out-of-sequence target is represented in the CFG by two or more arcs departing from the same vertex, where at least two of them are targeted to out-of-sequence vertices. These are said to be **multiple fan-out** blocks.

A **derived signature** is a value assigned to each instruction block. The term *derived* means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using such opcodes to fed a Linear Feed-Back Shift Register (LFSR). These values are calculated at compile time and used as reference by the EDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as *Embedded Signature Monitoring* (ESM). A **watchdog processor** is a hardware EDM used to detect *Control Flow Errors* (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. In this case it performs signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their references. If any difference is found the error in the main processor instruction stream is detected and an *Error Recovery Mechanism* (ERM) is activated.

3. Related work

Several hardware approaches using a watchdog processor and derived signatures for concurrent error detection have been proposed. The most relevant works are out-

lined below:

Ohlsson *et al.* present in [5] a watchdog processor built into a RISC processor. A specialized `tst` instruction is inserted in the delay slot of every branch instruction, testing the signature of the preceding block. An instruction counter is also used to time-out an instruction sequence when a branch instruction is not executed in the specified range. Other watchdog supporting instructions are added to the processor instruction set to save and restore the value of the instruction counter on procedure calls.

The watchdog processor used by Galla *et al.* in [8] to verify correct execution of a communications controller of the Time-Triggered Architecture uses a similar approach. A `check` instruction is inserted in appropiate places to trigger the checking process with the reference signature that is stored in the subsequent word. In the case of a branch, the branch delay slot is used to insert an adjustment value for the signature to ensure the run-time signature is the same at the check instruction independent of the path followed. An instruction counter is also used by the watchdog. The counter is loaded during the check instruction and decremented for every instruction executed; a time-out is issued if the counter reaches zero before a new check instruction is executed. Due to the nature of the communications architecture, no interrupts are processed by the controller. Thus saving the run-time signature or instruction counter is not necessary.

The ERC32 is a SPARC processor augmented with parity bits and a program flow control mechanism presented by Gaisler in [9]. In the ERC32 a test instruction to verify the processor control flow is also inserted in the delay slot of every branch to verify the instruction bits of the preceding block. In his work, the test instruction is a slightly modified version of the original `nop` instruction and no other modifications to the instruction set is needed.

A different error detection mechanism is presented by Kim and Somani in [10]. The decoded signals for the pipeline control are checked in a per instruction basis and their references are retrieved from a watchdog private cache. If the run-time signature of a given instruction can't be checked because its reference counterpart is not found in the cache, it is stored in the cache and used as reference for future executions. No signatures or program modifications are needed because reference signatures are generated at run-time, thus creating no overhead. The drawback in this approach is that the watchdog processor can't check all instructions. An instruction can be checked if it has been previously executed and only if its reference has not been displaced from the watchdog private cache to store other signatures. Although the error is detected before the instruction is committed and no overheads are created, the error coverage is poor.

More recently, hardware additions to modern processor architectures have been proposed to re-execute instructions and perform a comparison to verify no errors have been produced before instructions are committed.

Some of these proposals are outlined below for the sake of completeness but they are out of the scope of this work because: i) Hardware additions, spare components and/or time redundancy are used to detect all possible errors by re-execution of all instructions. Not only errors in the instruction bits or execution flow are detected but data errors as well. ii) They require either a complete redesign of the processor control unit or the addition of a complete execution unit capable to carry out the same set of instructions of the main processor, although its control unit can be simpler.

These include, to name a few:

- REESE (Nickel and Somani, [11]) and AR-SMT (Rotenberg, [12]). Both works take advantage of the simultaneous multi-threading architecture to execute every instruction twice. The instructions of the first thread, along with their operands and results are stored in a queue (a delay buffer in Rotenberg's work) and re-executed. Results of both executions are compared before the instructions are committed.
- The micropocessor design approach of Weaver and Austin in [13] to achive fault tolerance is the substitution of the committment stage of a pipeline processor with a checker processor. Instructions along with their inputs, addresses and the results obtained are passed to the checker processor where instructions are re-executed and results can be verified before they are committed.
- The O3RS design of Mendelson and Suri in [14] and a modified multiscalar architecture used by Rashid *et al.* in [15] use spare components in a processor capable of issuing more than one instruction per cycle to re-execute instructions.

4. System Architecture

The system (see Fig. 2) is built around a soft-core of a MIPS R3000 processor clone developed in synthesizeable VHDL [16]. It is a 5-stage pipelined RISC processor running the MIPS-I and MIPS-II Instruction Set Architecture [17]. Instruction and data bus interfaces are designed as AMBA [18] AHB bus masters providing external memory access.

This processor is provided with a Memory Management Unit (MMU) inside the System Control Coprocessor (*CP0* in the MIPS nomenclature) to perform virtual to physical address mapping, isolating memory areas of different processes and checking correct alignment of memory references.

To minimise performance penalty, the instruction cache is designed with two read ports that can provide two instructions simultaneously, one for each processor. On a cache hit, no interference exists even if the other processor is waiting for a cache line refill because of a cache miss.

To reduce the instruction cache complexity a single write port is provided that must be shared by both processors. When simulta-

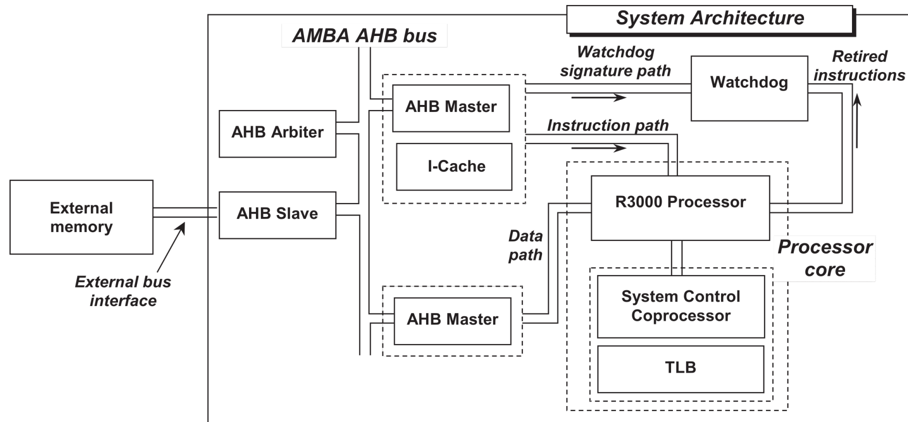

Figure 2: System architecture

neous cache misses happen, cache refills are served in a First-Come First-Served fashion. If they happen in the same clock cycle, the main processor is promoted.

This arrangement takes advantage of space locality in the application program to augment cache hits for signatures. As we use an ESM technique and signatures are interleaved with processor instructions, when both processors produce a cache miss they request the same memory block most of the times, as both reference words in the same program area.

No modification is needed in the processor instruction set due to the fact that signature instructions are neither fetched nor executed by the main processor. This allows us to maintain binary compatibility with existing software. If access to the source code is not possible, the program can be run without modification (and no concurrent flow error detection capability will be provided). This is possible because the watchdog processor and processor's modified architecture can be enabled and disabled under software control running with superuser privileges. If these features are disabled, our processor behaves as an off-the-shelf MIPS processor. Thus, if binary compatibility is needed for a given task, these features must be disabled by the OS every time the task resumes execution.

The watchdog processor is fed with the instructions from the main processor pipeline as they are retired. When these instructions enter the watchdog the run-time signatures and address parity bits are calculated at the same rate of the arrived instructions. When a block ends, these values are stored in a FIFO memory to decouple the signature checking process. This FIFO allows a large set of instructions to be retired from the pipeline while the watchdog is waiting for a cache refill in order to get a reference signature instruction. In a similar way, the FIFO can be emptied by the watchdog while the main processor pipeline is stalled due to a memory operation. When this memory is full, the pipeline if forced to wait for the watchdog checking process to read some data from the FIFO.

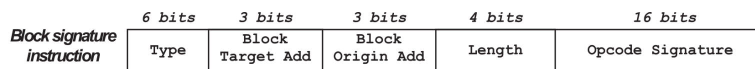

Figure 3: Block signature encoding

5. Interleaved Signature Instruction Stream

Block signatures are placed at the beginning of every basic block in our scheme. These reference signatures are used by the watchdog processor only and not processed in any way by the main processor.

Two completely independent, interleaved instruction streams coexist in our system: the application instruction stream which is divided into blocks and executed by the main processor and the signature stream used by the watchdog processor. We have called *Interleaved Signature Instruction Stream* (ISIS) to our technique due to this fact.

The signature word (see Fig. 3 for a field description) provide enough information to the watchdog processor to check the following block properties:

1. **Block length**. The watchdog processor checks a block's signature when the last instruction of the block is retired from the processor pipeline. Instead of relying on the branch instruction at the end of the block to perform the signature checking, the watchdog counts the instructions as they are retired. In this way, the watchdog can anticipate when the last instruction comes and detect a CFE if a branch occurs too early or too late.
2. **Block signature**. The block instructions are compacted using a 16-bit LFSR that will be used by the watchdog to verify that the correct instructions have been retired from the processor pipeline.
3. **Block Target Address**. In the case of a non multiple fan-out block with a target address that can be determined at compile-time, a 3-bit parity signature is computed from the address difference between the branch and the out-of-sequence target instruction. These parity bits are used at run-time to provide some confidence in that the instruction reached after the branch is the correct one.
4. **Block Origin Address**. When the branch of a multiple fan-out block is executed, the watchdog can't check all possible destinations even if they are obtainable at compile time. In our scheme, every possible destination block is provided with a 3-bit parity signature of the address difference between the originating branch and the start of the block, much the same as the previous Block Target Address check. Thus, instead of checking that the target instruction is the correct one, the watchdog processor checks (at the target block) that the originating branch is the correct one in this case.

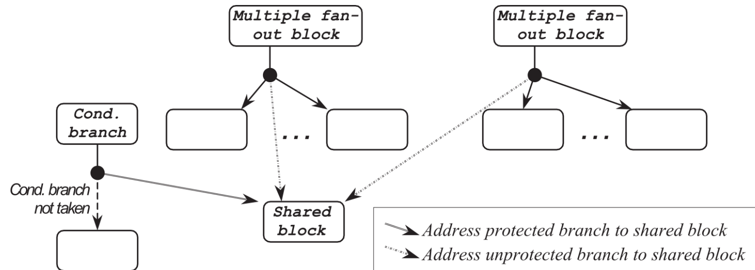

Figure 4: Example of an address checking uncovered case

The signature instruction encoding has been designed in such a way that a main processor instruction can not be misintrepreted as a watchdog signature instruction. This provides an additional check when a branch instruction is executed by the main processor. This check consists in the requirement to find a signature instruction immediately preceding the first instruction of every block. This also helps to detect a CFE if a branch erroneously reaches a signature instruction, because the used encoding will force an illegal instruction exception to be raised.

Furthermore, the block type helps the watchdog processor to check wether the execution flow is correct. For example, in the case of a multiple fan-out block the block type reflects the need to check the address signature at the target block. Even if an incorrect branch is taken to the initial instruction of a block, target's signature instruction must have coded in its type that it is a block where the origin address must be checked or a CFE exception will be raised.

Instructions in the MIPS processor must be placed at word boundaries; a memory alignment exception is raised if this requirement is not met. Taking advantage of this mechanism, the watchdog processor computes address differences as 30-bit values. Given that the branch instruction type used most of the time by the compiler use a 16-bit offset to reach the target instruction these differences obtained at run-time for Block Target Address and Block Origin Address checks are usually half empty, so every parity bit protects 5 (10 in the worst case) of such bits.

To our knowledge, the Block Origin Address checking has never been proposed in the literature. The solutions offered so far to manage jumps with multiple targets use *justifying signatures* (see [7] for an example) to patch the run-time signature and delay the check process until a common branch-in point is encountered, increasing the error detection latency.

Not all jumps can be covered with address checking however. Neither the jumps with run-time computed addresses nor those jumps to a multiple fan-in block that it is shared by several multiple fan-out blocks (see Fig. 4 for an example). In the later case, an address signature per origin should be used in the fan-in block, which is not possible. Currently, only Block Origin Address checks from non multiple fan-out blocks can

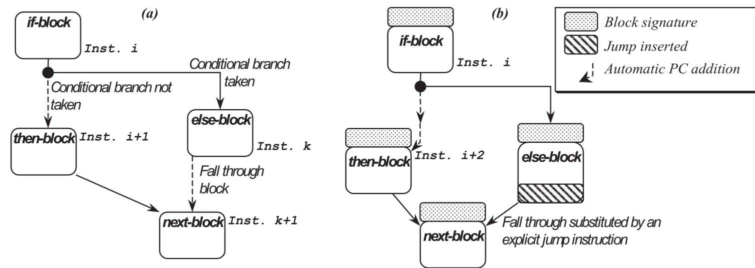

Figure 5: An `if-then-else` example (a). After block signatures and jump insertion (b)

be covered for such shared blocks.

5.1. Processor Architecture Modifications

Isolating the reference signatures from the instructions fed into the processor pipeline results in a minimal performance overhead in the application program. Sligth architecture modifications are needed in the main processor in order to achieve it.

First of all, when a conditional branch instruction ends a basic block, a second block follows immediately. The second block's signature sits between them, and the main processor must skip it. In order to effectively jumping over the signature, the signature size is added to the Program Counter if the branch is not taken.

In the same way, when a procedure call instruction ends a basic block the next one to be executed after the procedure returns immediately follows the first one. Again, the second block's signature must be taken into account when calculating the procedure return address. And again, this is achieved by an automatic addition of the signature size to the PC.

Additions to the PC mentioned above can be automatically generated at run-time because the control unit decodes a branch or procedure call instruction at the end of the block. The instruction is a clear indication that the block end will arrive soon. As the processor has a pipelined architecture, the next instruction is executed in all cases (this is known as the *branch delay slot*), so the control unit has a clock cycle to prepare for the addtion. Despite the fact that the instruction in the delay slot is placed after the branch, it logically belongs to the same block, as it is executed even if the branch is taken.

However, in the case of a block fall-through the control unit has no clue to determine when the first block ends, so the signature can not be automatically jumped over. In this case, the compiler explicitly adds an unconditional jump to skip it. This is the only case where a processor instruction must be added in order to isolate main processor from the signature stream. Figure 5a shows an example of an `if-then-else` construct with a fall-through block that needs such an addition (shown in Fig. 5b).

6. Overhead analysis

Although we have not enough experimental data yet to assess the memory and performance overhead of our system, a qualitative analysis for the memory overhead based on related work is possible.

A purely software approach to concurrent error detection was evaluated by Wildner in [19]. This control flow EDM is called *Compiler-Assisted Self Checking of Structural Integrity* (CASC) and it is based on address hashing and signature justifying to protect the return address of every procedure. At the procedure entry, the return address of the procedure is extracted from the link register into a general-purpose register to be operated on. The first operation is the inversion of the LSB bit of the return address to provide a misaligment exception in the case of a CFE. An `add` instruction at each basic block is inserted to justify the procedure signature and, at the exit point, the final justifying and reinversion of the LSB bit is calculated and the result is transferred to the link register before returning from the procedure. In the case of a CFE, the return address obtained is likely to cause a misaligment exception thus catching the error. The experiments carried out on a RISC SPARC processor resulted in a memory codesize overhead for the SPECint92 benchmarks varying from 0% to 28% (18,76% on average) depending on the run-time library used and the benchmark itself.

The hardware watchdog of Ohlsson *et al.* presented in [5] use a `tst` instruction per basic block, taking advantage of the branch delay slot of a pipelined RISC processor called TRIP. One of the detection mechanisms used by the watchdog is an instruction counter to issue a time-out exception if a branch instruction is not executed during the specified interval. When a procedure is called two instructions are inserted to save the block instruction counter and another instrucion is inserted at the procedure end to restore it. Their watchdog code size overhead is evaluated to be between 13% and 25%. The later value comes from the heap sort algorithm showing a mean basic block of 4.8 instructions.

ISIS inserts a single word per basic block, without special treatment for procedure entry and exit blocks, so CASC or TRIP overhead can be taken as an upper bound of ISIS memory overhead.

Hennessey and Patterson in [20] state that the average length of a basic block for a RISC processor sits between 7 and 8 instructions. The reasoning to evaluate memory overhead as $1/L$ being L the basic block length is used by Ohlsson and Rimén in [21] to evaluate the memory overhead of their *Implicit Signature Checking* (ISC) method. The same value (7-8 instructions per block) is used by Shirvani and McCluskey in [22] to perform this same analysis on several software signature checking techniques.

Applying this evaluation method to ISIS results in a mean of about 12% - 15% memory overhead. An additional word must be accounted to eliminate fall-through blocks. The overhead of these insertions has to be methodically studied, but initial exper-

iments show a negligible impact on overall memory overhead.

7. Conclusion

We have presented a novel technique to embed signatures into the execution flow of a RISC processor that provides a set of error checking procedures to assess that the flow of executed instructions is correct. These checking procedures include a block length count, the signature of instruction opcodes using a LFSR, and address checking when a branch is executed. All these checkings are performed in a per block basis, in order to reduce the error detection latency of our hardware Error Detection Mechanism.

One of those address checking procedures has not been published before. It is the Block Origin Address checking used when a branch has multiple valid targets and consists of delaying the branch checking until the target instruction is reached and verifying that the branch comes from the correct origin vertex in the CFG. This technique solves the address checking problem that arises if a branch has multiple valid destinations, for example, the table-based jumps used when the OS dispatchs a service request.

Not all software cases can be covered with address checking however. When a CFG vertex is targeted from two or more multiple fan-out vertices the Block Origin Address check becomes ineffective.

We have called *Interleaved Signature Instruction Stream* (ISIS) to our signature embedding technique to reflect the important fact that signature instructions processed by the watchdog processor and main processor instructions are two completely independent streams.

ISIS has been implemented into a RISC processor and the modifications demanded by signature embedding to the original architecture have been discussed. These modifications are very simple and can be enabled and disabled by software with superuser privileges to maintain binary compatibility with existing software. No specific features of the processor has been used, so the port of ISIS to a different processor architecure is quite straightforward.

Memory performance overhead has been studied by comparison with other methods and analysis show a memory overhead between 12% and 15% although we haven't performed a methodical study yet. As a negligible amount of instructions are added to the original program, the performance is expected to remain basically unaltered.

Acknowledgements

This work is supported by the Spanish Government *Comisión Interministerial de Ciencia y Tecnología* under project CICYT TAP99-0443-C05-02.

Bibliography

[1] Avizienis, A.: Building Dependable Systems: How to Keep Up with Complexity. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 4-14, Pasadena, California, 1995.

[2] Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. *Proc. of the 19th Fault Tolerant Computing Symposium (FTCS-19)*, 340-347, Chicago, Illinois, 1989.

[3] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[4] Gaisler, J.: Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-25)*, 42-46, Seattle, Washington, 1997.

[5] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[6] Siewiorek, D.P.: Niche Sucesses to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 26-33, Pasadena, California, 1995.

[7] Oh, N., Shirvani, P.P., McCluskey, E.J.: Control Flow Checking by Software Signatures. *IEEE Transactions on Reliability - Special Section on Fault Tolerant VLSI Systems*, March, 2001.

[8] Galla, T.M., Sprachmann, M., Steininger, A., Temple, C.: Control Flow Monitoring for a Time-Triggered Communication Controller. *Proceedings of the 10th European Workshop on Dependable Computing (EWDC-10)*, 43-48, Vienna, Austria, 1999.

[9] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[10] Kim, S., Somani, A.K.: On-Line Integrity Monitoring of Microprocessor Control Logic. *Proc. Intl. Conference on Computer Design: VLSI in Computers and Processors (ICCD-01)*, 314-319, Austin, Texas, 2001.

[11] Nickel, J.B., Somani, A.K.: REESE: A Method of Soft Error Detection in Microprocessors. *Proc. of the 2001 Intl. Conference on Dependable Systems and Networks (DSN-2001)*, 401-410, Goteborg, Sweden, 2001.

[12] Rotenberg, E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *Proc. of the 29th Fault Tolerant Computing Symposium (FTCS-29)*, 84-91, Madison, Wisconsin, 1999.

[13] Weaver, C., Austin, T.: A Fault Tolerant Approach to Microprocessor Design. *Proc. of the 2001 Intl. Conference on Dependable Systems and Networks (DSN-2001)*, 411-420, Goteborg, Sweden, 2001.

[14] Mendelson, A., Suri, N.: Designing High-Performance & Reliable Superscalar Architectures. The Out of Order Reliable Superscalar (O3RS) Approach. *Proc. of the 2000 Intl. Conference on Dependable Systems and Networks (DSN-2000)*, 473-481, New York, USA, 2000.

[15] Rashid, F., Saluja, K.K., Ramanathan, P.: Fault Tolerance Through Reexecution in Multiscalar Architecture. *Proc. of the 2000 Intl. Conference on Dependable Systems and Networks (DSN-2000)*, 482-491, New York, USA, 2000.

[16] *IEEE Std. 1076-1993: VHDL Language Reference Manual.* The Institute of Electrical and Electronics Engineers Inc., New York, 1995.

[17] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture.* MIPS Technologies, 2001.

[18] *AMBA Specification rev2.0.* ARM Limited, 1999.

[19] Wildner, U.: Experimental Evaluation of Assigned Signature Checking With Return Address Hashing on Different Platforms. *Proc. of the 6th Intl. Working Conference on Dependable Computing for Critical Applications*, 1-16, Grainau, Germany, 1997.

[20] Hennessy, J.L., Patterson, D.A.: *Computer Architecture. A Quantitative Approach*, 2nd edition, Morgan-Kauffmann Pub., Inc., 1996.

[21] Ohlsson, J., Rimén, M.: Implicit Signature Checking. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 218-227, Pasadena, California, 1995.

[22] Shirvani, P.P., McCluskey, E.J.: Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project. *Center for Reliable Computing*, Technical Report CRC-98-2, Standford, California, 1998.

The HORUS Processor

F. Rodríguez, J. C. Campelo and J. J. Serrano

Grupo de Sistemas Tolerantes a Fallos - *Fault Tolerant Systems Group*
Dept. Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, 46022 Valencia (Spain),
e-mail: {prodrig, jcampelo, jserrano}@disca.upv.es,
http://www.disca.upv.es/gstf

Abstract

Designing a complete SoC or reuse SoC components to create a complete system is a common task nowadays. The flexibility offered by the design flow used offers the designer an unprecedented capability to incorporate more and more demanded features like error detection and correction mechanisms to increase the system dependability. This paper describes the design of the HORUS processor, a RISC processor augmented with a concurrent error mechanism and the architectural modifications needed on the original design. Taking advantage of modern high-level design methodology and using the VHDL modeling language, the standard architecture has been slightly modified to minimize the resulting performance penalty.

1. Introduction

With the advent of modern technologies in the field of programmable devices and enormous advances in the software tools used to model, simulate and translate into real hardware almost any complex digital system, the capability to design a whole System-On-Chip (SoC) has become a reality even for small companies. With the widespread use of embedded systems in our everyday life, service availability and dependability concerns for these systems are increasingly important [1].

A SoC is usually modelled using a Hardware Description Language (HDL) like VHDL [2]. It allows a hierarchical description of the system and the designed elements interconnect much the same way as they would in a graphical design flow, but using an arbitrary abstraction level. It also provides IO facilities to easily incorporate test vectors, and language assertions to verify the

correct behaviour of the model during the simulation.

Efficient error detection is of fundamental importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent *Error Detection Mechanism* (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experiments demonstrate [3, 4, 5, 6], a high percentage of non-overwritten errors results in control flow errors.

The possibility to modify the original architecture of a processor modelled using VHDL gives the SoC designer an unprecedented capability to incorporate EDM's which were previously available at large design companies only.

Siewiorek states in [7] that "To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users". A fault-tolerant technique can be considered transparent only if results in minimal performance overhead in silicon, memory size or processor speed. Although redundant systems can achieve the best degree of fault-tolerance, the high overheads imposed limit their applicability in every day computing elements. The same limitation applies when a software only solution is used, due to the processor's performance penalty incurred.

Siewiorek's statement can be also translated into the SoC world, to demand fault-tolerant techniques that minimise their impact on performance (the scarcest resource in such systems) if we want those techniques to be used at all.

The work presented here describes the design of the HORUS processor. It is a classic pipelined RISC processor designed in VHDL that has been augmented with a concurrent error detection mechanism of control flow errors with no performance penalty and minimal memory and silicon sizes. No modifications are needed in the instruction set of the processor used and the architectural ones are so small that they can be enabled and disabled under software control to allow binary compatibility with existing software. The watchdog processor is very simple, and its design can be applied to other RISC processors as well.

The paper is structured as follows: The next section is devoted to present a set of basic definitions and it is followed by the outline of related works in the literature. Section 4 presents the signature embedding technique the system uses. Section 5 discusses the processor architecture and its compiler support. A memory overhead comparison with similar work is performed afterwards, which is followed by some preliminary synthesis results to finish with the conclusions.

2. Basic terms

A computer program can be represented by a *Control Flow Graph* (CFG). Vertices in this graph are used to represent basic blocks and directed arcs are used to represent legal paths between blocks.

A *basic block* is a sequence of instructions to be executed in order, with no branch targets except for the very first instruction and with no branch instructions except possibly the last one (if any).

A *derived signature* is a value assigned to each instruction block. The term *derived* means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using such opcodes to fed a Linear Feed-Back Shift Register (LFSR). These values are calculated at compile time and used as reference by the EDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as *Embedded Signature Monitoring* (ESM). A *watchdog processor* is an EDM hardware device used to detect *Control Flow Errors* (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. In this case it performs signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their references. If any difference is found the error in the main processor instruction stream is detected and an *Error Recovery Mechanism* (ERM) is activated.

3. Related work

Several approaches using an ESM watchdog processor and derived signatures for concurrent error detection have been proposed in the literature. The most relevant works are outlined below. Other recent proposals in the concurrent error detection field are based in last generation processor features like support for simultaneous multithreading or superscalar architectures. These approaches are valid, but of very limited use in SoC designs however, due to the simpler architectures these processors are based on.

Ohlsson *et al* present in [6] a watchdog processor built into a RISC processor. A specialized `tst` instruction is inserted in the delay slot of every branch instruction, testing the signature of the preceding block. An instruction counter is also used to time-out the instructions sequence when a branch instruction is not executed in the specified range. Other watchdog supporting instructions are added to the processor instruction set to save and restore the value of the instruction counter on procedure calls.

The watchdog processor used by Galla *et al* in [8] to verify correct execution of a communications controller of the Time-Triggered Architecture uses a similar approach. A `check` instruction is inserted in appropriate places to trigger the checking process with the reference signature that is stored in the subsequent word. In the case of a branch, the branch delay slot is used to insert an adjustment value for the signature to ensure the run-time signature is the same at the check instruction independent of the

path followed. The watchdog also uses an instruction counter. This is loaded during the check instruction and decremented for every instruction executed; a time-out is issued if the counter reaches zero before a new check instruction is executed. Due to the nature of the communications architecture, no interrupts are processed by the controller, thus saving the run-time signature or instruction counter is not necessary.

The ERC32 is a SPARC processor augmented with parity bits and a program flow control mechanism presented by Gaisler in [9]. In the ERC32 a test instruction to verify the processor control flow is also inserted in the delay slot of every branch to verify the instruction bits of the preceding block. In his work, the test instruction is a slightly modified version of the original `nop` instruction and no other modifications to the instruction set is needed.

4. ISIS: Interleaved Signatures Instruction Stream

All ESM watchdogs presented in the preceding section require processor cycles to check instruction signatures so a performance penalty results inevitable. As the length of block sequences of a RISC processor is between 4 and 10 instructions, the memory and performance overhead is quite noticeable.

To reduce performance overhead, the scarcest resource when targeting to field-programmable devices, the main CPU should not process signatures in any way. With this objective in mind, we have designed a CPU that skips an instruction per basic block while maintaining CPU instruction sequencing.

Those instruction gaps are filled with block signatures and processed by the watchdog processor. With this arrangement, two completely independent interleaved instruction streams coexist in our system: the application instruction stream, which is divided into blocks and executed by the main processor and the signature stream, used by the watchdog processor. We have called *Interleaved Signature Instruction Stream* (ISIS) to our technique due to this fact. More information about the error detection mechanisms included with the block signature can be found in [10].

Isolating the reference signatures from the instructions fed into the processor pipeline results in a minimal performance overhead in the application program. To achieve this isolation, several architecture modifications are needed in the main processor. These are summarized below:

a) When a conditional branch instruction ends a basic block, a second block follows immediately. The second block's signature sits between them, and the main processor must skip it. In order to effectively jumping over the signature, the signature size is added to the Program Counter if the branch is not taken.

b) When a procedure call is executed, the block to be executed after returning

the procedure immediately follows the first one. Again, the second block's signature must be taken into account when calculating the procedure return address. And again, this is achieved by an automatic addition of the signature size to the PC when the procedure is called.

c) In the preceding cases, the processor can perform PC additions when executes a branch or call instruction. However, when a fall-through block is executed, the processor has no way to determine when the last instruction of the block arrives. A *fall-through block* is a basic block that does not end with a branch instruction; that is, it ends because the instruction that follows the block is the target of a branch. To help the CPU, the compiler inserts a jump instruction to signal the end of the block. This is the only case where a processor instruction must be added in order to isolate main processor from the signature stream. Figure 1 shows an example of an `if-then-else` construct with such a block and Fig. 2 the same construct after the jump instruction and the signatures have been added.

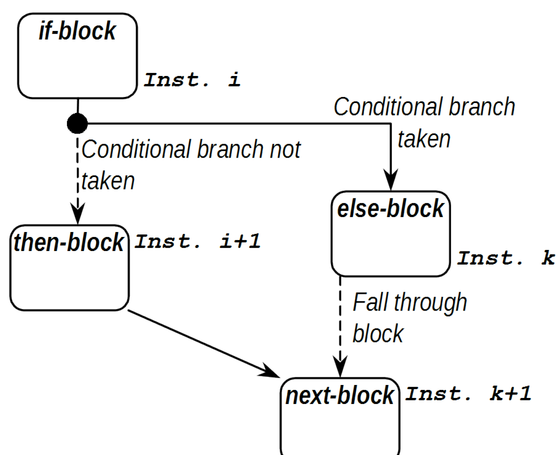

Figure 1: If-then-else construct with a fall-through block

5. The HORUS processor

The system (see Fig. 3) is built around the HORUS processor, a soft-core of a MIPS R3000 processor clone developed in synthesizeable VHDL. It is a four stage pipelined RISC processor running the MIPS-I and MIPS-II Instruction Set Architecture [11] except that it does not provide floating point support. It has been augmented to include a watchdog processor implementing the ISIS technique. Instruction and data

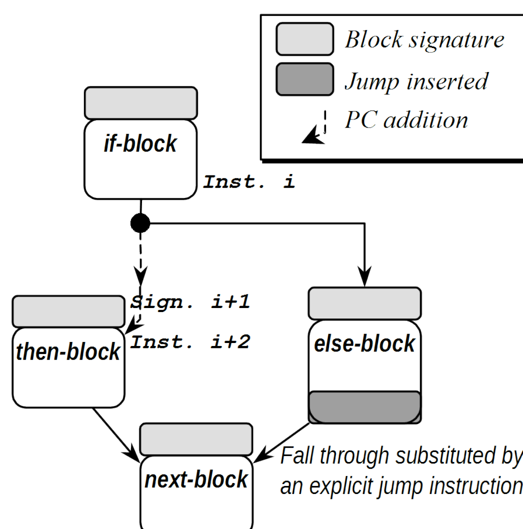

Figure 2: If-then-else construct after signatures and jump instruction inserted

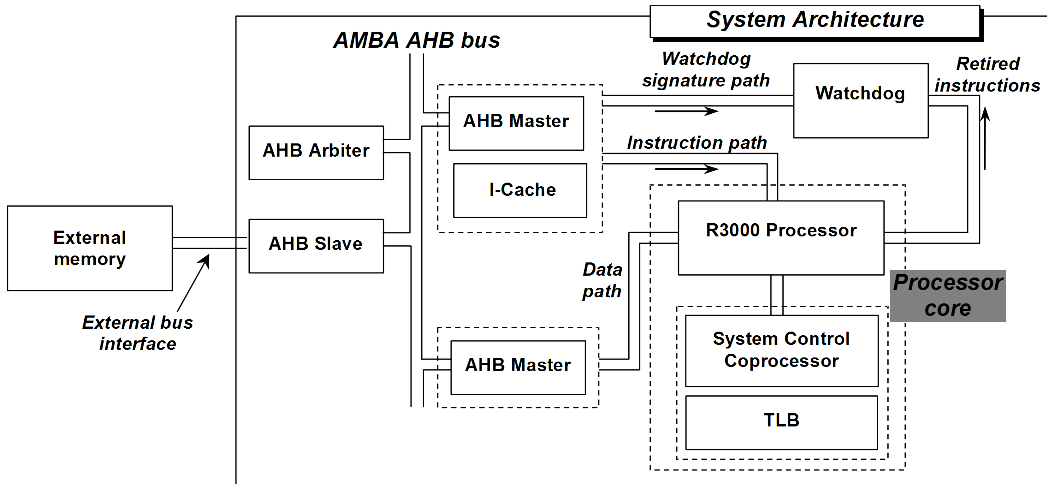

Figure 3: HORUS processor and overall system architecture

bus interfaces are designed as AMBA [12] AHB bus masters providing memory access for main and watchdog processors.

This processor is provided with a Memory Management Unit (MMU) to perform virtual to physical address mapping, isolating memory areas of different processes and checking correct alignment of memory references.

To minimize performance penalty, the instruction cache is designed with two read ports that can provide two instructions simultaneously, one for each processor. On a cache hit, no interference exists even if the other processor is waiting for a cache line refill because of a cache miss.

To reduce the instruction cache complexity a single write port is provided that must be shared by both processors. When simultaneous cache misses happen, cache refills are served in a First-Come First-Served fashion. If they happen in the same clock cycle, the main processor is promoted.

This arrangement takes advantage of space locality in the application program to augment cache hits for signatures. As we use an ESM technique and signatures are interleaved with processor instructions, when both processors produce a cache miss they request the same memory block most of the times, as both reference words in the same program area with very few cycles of difference.

No modification is needed in the processor instruction set due to the fact that signature instructions are neither fetched nor executed by the main processor. This allows us to maintain binary compatibility with existing software. If access to the source code is not possible, the program can be run without modification (and no concurrent error detection capability will be provided).

This is possible because the watchdog processor and processor's modified architecture can be enabled and disabled under software

control running with superuser privileges. If these features are disabled, our processor behaves as an off-the-shelf MIPS processor. Thus, if binary compatibility is needed for a given task, the OS must disable these features every time the task resumes execution.

The watchdog processor is fed with the instructions from the main processor pipeline as they are retired. When these instructions enter the watchdog the run-time signatures are calculated at the same rate of the arrived instructions. When a block ends, these values are stored in a FIFO memory to decouple the signature checking process.

This FIFO allows a large set of instructions to be retired from the pipeline while the watchdog is waiting for a cache refill in order to get a reference signature instruction. In a similar way, the watchdog can empty the FIFO while the main processor pipeline is stalled due to a memory operation. When this memory is full, the pipeline if forced to wait for the watchdog checking process to read some data from the FIFO.

5.1. HORUS compiler support

The GNU `gcc` compiler already provided a port to target MIPS processors. As its source code is freely available it was the natural starting point to provide the required software support for the HORUS processor. The `gas` program (GNU Assembler) has the responsibility of the assembly stage in the compilation process, after program optimization passes and before the final linker stage.

Gas has been modified to support the HORUS MIPS modified architecture and its optional use of the ISIS technique via command line switches.

As instructions are assembled,

1. If the current instruction is the target of a branch instruction, a new block must start and so its signature must be inserted.
2. If the current instruction is a branch, the next instruction will fill the branch delay slot and end the current block.

With this information and the opcode bits of the program instructions the assembler can calculate block signatures and insert them at appropriate places.

No provisions are needed to modify the target of a branch or call instruction, as all instruction addresses are referred using symbolic names (labels).

6. System comparison

In this section the HORUS processor is compared against the most related works in the literature, those of Ohlsson *et al* presented in [6] and Gaisler presented in [9].

Memory requirements for block signatures are fundamentally the same as Gaisler's processor [5]. His processor uses a test instruction to fill every branch delay slot. That is, a test instruction is used for every block exactly the same as the signature

instruction is used for every block in the HORUS processor.

The system architecture of Ohlsson et al's work requires additional instructions at procedure entry and exit points, so its memory requirements are larger.

The main differences are in the processor cycles used by the error detection mechanisms. While the HORUS processor demands no CPU cycles to process block signatures, Gaisler's one wastes one cycle per block and Galla's requires additional cycles to process procedures. These differences reflect the fact that signatures are not instructions for the HORUS processor.

The HORUS performance is however affected by the watchdog processor.

First, it shares the instruction memory through the cache controller with the main processor. As both processors reference the same space locations at approximately the same time (with a few clock cycles of difference) the instruction cache contains the block signature referenced by the watchdog processor most of the times.

Second, to solve the problems arose with fall-through blocks the compiler inserts a jump instruction in the processor instruction stream. Some figures about the memory consumption by signatures in the HORUS processor are presented in [10]. Results show the memory overhead (15 % to 28 %) is comparable to Ohlsson's TRIP processor (13 % to 25 %).

6.1. Logic synthesis

Preliminary synthesis results show a silicon overhead about 8,5 %. This has been obtained comparing the synthesis results from two different versions: the initial design (without watchdog processor) and the whole system previously described in this paper.

The device used for this synthesis has been the Virtex2 xc2v6000-4 from Xilinx. This device has been selected because it offers a large pool of logic resources and routing. This way, the overall equivalent gate count provided by the synthesis tools is dominated by the logic used not the routing resources. This equivalent gate count provides an overall complexity mark. This value must be taken with caution, however, as it is a global estimation made by the software tool from the different logic resources used: logic cells, distributed memory cells, routing elements, and dedicated memory blocks.

Some technical details must be explained to understand this result. Firstly, the instruction cache and the MMU provide a single port that is multiplexed to provide data to main and watchdog processors. This multiplexing circuitry creates two "virtual" access ports, one for the main processor and the second one for the watchdog processor. It works at twice the frequency of the main processor and so does not imply any additional delay cycles for the processor. This way, the silicon overhead for the inclusion of the watchdog processor is reduced, as it accounts for the multiplexing circuitry only. The difference between the initial design

and the design augmented by the watchdog processor can be roughly considered to be these multiplexing circuits and the watchdog processor itself.

Secondly, large logic blocks originally designed with a goal of technology independence have been redesigned to take advantage of specific macro blocks offered by the tool vendor (Xilinx). These blocks include the content addressable memories (CAMs) used by the instruction cache and the MMU, and the FIFO memories inside the watchdog processor used to connect the signature run-time calculation and reference checking processes.

7. Conclusions

The architecture of the HORUS processor has been presented. It is a RISC pipelined processor augmented with a concurrent error detection mechanism (a watchdog processor) specifically designed to minimize the CPU performance penalty. This goal is of primary importance to successfully employ fault tolerance mechanisms in the current market of SoC systems, where the clock frequency when targeting field programmable devices is far from the general-purpose personal computer ranges.

The performance overhead reduction is achieved modifying the standard meaning of branch and call instructions. This is possible when architecture modifications are possible; increasingly larger devices, modern design methodologies and its tools now offer this possibility to the designer. Compiler support for this architecture has been also outlined to be of reasonable complexity.

Finally, a light comparison with similar propositions has been carried out. While memory requirements are fundamentally the same as previous proposals, performance benefits from the fact that signatures are neither fetched nor processed by the CPU.

The HORUS processor has been designed using a synthesizeable subset of the VHDL language and it is currently under beta test. Preliminary results show the logic needed in include the watchdog processor increases the system size by a moderate 8,5 % factor. Characterization of the watchdog processor is currently underway.

Acknowledgements

This work is supported by the Spanish Government *Comisión Interministerial de Ciencia y Tecnología* under project CICYT TAP99-0443-C05-02.

Bibliography

[1] Avresky, D., Grosspietsch, K.E., Jhonson, D.W., and Lombardi, F.: Embedded Fault Tolerant Systems. *IEEE Micro Magazine*, 8-11, Vol. 18, No. 5, 1998.

[2] *IEEE Std. 1076-1993: VHDL Language Reference Manual.* The Institute of Electrical and Electronics Engineers Inc., New York, 1995.

[3] Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of Error Detection Schemes Using Fault

Injection by Heavy-ion Radiation. *Proc. of the 19th Fault Tolerant Computing Symposium (FTCS-19)*, 340-347, Chicago, Illinois, 1989.

[4] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[5] Gaisler, J.: Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-27)*, 42-46, Seattle, Washington, 1997.

[6] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[7] Siewiorek, D.P.: Niche Sucesses to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 26-33, Pasadena, California, 1995.

[8] Galla, T.M., Sprachmann, M., Steininger, A., Temple, C.: Control Flow Monitoring for a Time-Triggered Communication Controller. *Proceedings of the 10th European Workshop on Dependable Computing (EWDC-10)*, 43-48, Vienna, Austria, 1999.

[9] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[10] Rodríguez, F., Campelo, J.C, Serrano, J.J.: A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream. to be presented at the *IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'2002)*, Vancouver, Canada, Nov. 2002.

[11] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture.* MIPS Technologies, 2001.

[12] *AMBA Specification rev2.0.* ARM Limited, 1999.

Delivering Error Detection Capabilities into a Field Programmable Device: The HORUS Processor Case Study

F. Rodríguez, J. C. Campelo, J. J. Serrano

Dept. Informática de Sistemas y Computadores,
Universidad Politécnica de Valencia
e-mail: {prodrig, jcampelo, jserrano}@disca.upv.es

Abstract

Designing a complete SoC or reuse SoC components to create a complete system is a common task nowadays. The flexibility offered by current design flows offers the designer an unprecedented capability to incorporate more and more demanded features like error detection and correction mechanisms to increase the system dependability. This is especially true for programmable devices, were rapid design and implementation methodologies are coupled with testing environments that are easily generated and used. This paper describes the design of the HORUS processor, a RISC processor augmented with a concurrent error mechanism, the architectural modifications needed on the original design to minimize the resulting performance penalty.

1. Introduction

With the advent of modern technologies in the field of programmable devices and enormous advances in the software tools used to model, simulate and translate into real hardware almost any complex digital system, the capability to design a whole System-On-Chip (SoC) has become a reality even for small companies. With the widespread use of embedded systems in our everyday life, service availability and dependability concerns for these systems are increasingly important [1].

Efficient error detection is of fundamental importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent Error Detection Mechanism (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experi-

ments demonstrate [2, 3], a high percentage of non-overwritten errors results in control flow errors.

The possibility to modify the original architecture of a processor modelled using a language like VHDL gives the SoC designer an unprecedented capability to incorporate EDM's which were previously available at large design companies only.

Siewiorek states in [4] that "To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users". A fault-tolerant technique can be considered transparent only if results in minimal overhead in silicon, memory size or processor performance. Although redundant systems can achieve the best degree of fault-tolerance, the high overheads implied limit their applicability in field programmable devices, were silicon is the scarcest resource. The same limitation applies when a software only solution is used, due to the processor's performance penalty incurred.

Siewiorek's statement can be also interpreted as a demand to implement fault-tolerant techniques that minimise their impact on the main processor clock if we want those techniques to be used at all.

The work presented here describes the overheads incurred in the HORUS processor [5] to incorporate a concurrent EDM. HORUS is a classic pipelined RISC processor designed in VHDL and synthesized into a Xilinx FPGA that has been augmented with a concurrent EDM of control flow errors with minimal performance and silicon penalty and moderate memory overhead.

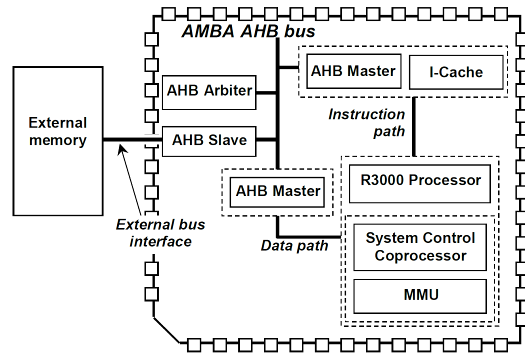

Figure 1: Original system architecture.

The paper is structured as follows: The next section is devoted to present the overall system architecture, including the watchdog processor and its connection with the rest of the system. Section 3 presents the overhead results of our approach, taking into account the memory footprint, performance loss and silicon complexity, to finish with the conclusions.

2. Introduction of the HORUS processor architecture

The original processor architecture (see Figure 1) is built around a soft-core of a MIPS R3000 processor clone developed in synthesizeable VHDL. It is a four stage pipelined RISC processor running the MIPS-I and MIPS-II Instruction Set Architecture except that it does not provide floating point support. Instruction and data bus interfaces are designed as ARM's AMBA AHB bus masters providing memory access for main and watchdog processors.

This processor is provided with a Memory Management Unit (MMU) to perform virtual to physical address mapping, isolating memory areas of different processes and checking correct alignment of memory references.

2.1. Error detection mechanism

The main processor has been augmented to include a watchdog processor implementing the ISIS (Interleaved Signature Instruction Stream) technique [5]. This watchdog processor is capable of detecting control flow errors and instruction errors in the main processor.

The instructions for this watchdog processor (called signatures) are interleaved with the instructions of the main processor. However, the main processor automatically skips them and conversely, the watchdog processor fetch signatures only and jumps over the main processor instructions.

No modification is needed in the processor instruction set due to the fact that watchdog instructions are neither fetched nor executed by the main processor. This allows us to maintain binary compatibility with existing software. If access to the source code is not possible, the program can be run without modification (and no concurrent error detection capability will be provided).

The modifications to the original processor architecture are described in detail in [5]. These can be enabled and disabled under software. If these features are disabled, our

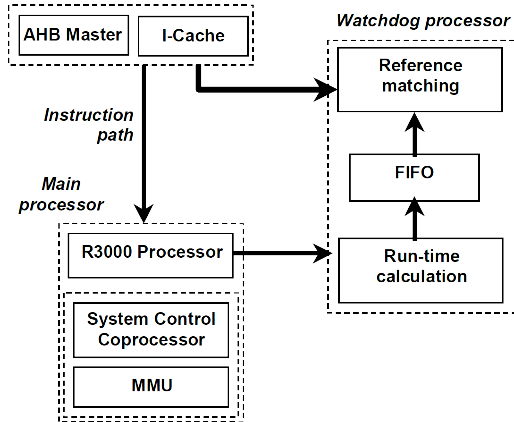

Figure 2: Augmenting the system with the watchdog processor.

processor behaves as an off-the-shelf MIPS processor. Thus, if binary compatibility is needed for a given task, the OS must disable these features every time the task resumes execution.

To minimize the resulting performance penalty, the instruction cache is designed with two read ports that can provide two instructions simultaneously, one for the main processor and one for the watchdog. If the processor gets a cache hit, no interference exists even if the other processor is stalled because of a cache miss.

This arrangement takes advantage of space locality in the application program to augment cache hits for signatures. As signatures are interleaved with processor instructions, when both processors produce a cache miss they request the same memory block most of the times, as both reference words in the same program area with very few cycles of difference.

To detect errors, the watchdog processor is

fed with the instructions from the main processor as they are retired from its pipeline (see Figure 2). The main processor instructions are treated as data by the watchdog, and are processed at their arrival rate. Every time the main processor executes a branch instruction, the watchdog processor checks the main processor execution flow comparing a run-time calculated value with its reference (the signature). If a mismatch is encountered, an exception is raised to signal the error detection.

As the watchdog processor needs some memory data (the signature) to perform the check, it can not be guaranteed that it is ready by the time the processor executes the branch. To decouple the checking process, a FIFO memory is used to store the run-time calculated values while the signature is fetched. This FIFO allows a large set of instructions to be retired from the main processor pipeline while the watchdog is waiting for the signature (due to a cache miss). In a similar way, the watchdog can empty the FIFO while the main processor pipeline is stalled due to a memory operation.

3. Overhead analysis

The next subsections are devoted to present the overhead related to the inclusion of the watchdog processor in the system design. This analysis includes memory, performance, clock frequency and silicon.

3.1. Memory overhead

The table 1 below shows some measurements obtained using a modified version of the gnu compiler. The data produced include the number of instructions of the original program, the number of instructions inserted to allow the main processor jump over signatures and the number of signatures.

These programs were selected as a representation of the type of programs (iterative, recursive, mixed) expected to be run into these systems.

fft. It is the Fast Fourier Transform applied to a random set of values. It is a good representation of a sequential program, as it has a single block with 168 (!) sequential instructions.

hanoi. This is the classic programming problem of hanoi towers.

quicksort. This program sorts a randomly initialized array of numbers. It is a mix of sequential statements and recursion as the specific version extensively uses unrolled loops and function inlining.

matrix. This program solves an integer matrix multiply.

queens. It solves the classic nine queens placement problem.

Although memory overhead may seem large at first sight, comparison of these data in [6] demonstrates they are similar to the overhead obtained with previous published watchdogs.

Table 1: Memory overhead

Program	Original instr.	Inserted instr.	Signatures added	Overhead
Fft	281	11	33	15,66 %
Hanoi	118	0	23	19,50 %
Quicksort	844	36	203	28,32 %
Matrix	139	5	27	23,02 %
Queens	305	11	59	22,95 %

3.2. Performance overhead

To obtain the performance overhead we have simulated the VHDL model of the FPGA device and directed it to execute the test programs presented in the previous section. Two versions of each program have been simulated, the original one (with no signatures, simulated with the watchdog processor disabled) and the version augmented with signatures for the watchdog processor (simulated with the watchdog processor enabled).

The table 2 shows the number of CPU cycles needed to complete the execution of the test programs. It is evident from these data that the main processor sees no appreciable sign of degraded performance when the watchdog processor is in use.

With a memory overhead above 19 %, the performance result from the hanoi test program seems incredible. However, this result can be explained if we take into account that the main processor is always promoted when simultaneous cache misses occur. Analyzing the simulation trace in detail, results shown that the main processor was always executing a set of instructions while the watchdog processor was waiting for the signature word to check precisely those instructions. That is, the main processor was always several instructions ahead the checking process of the watchdog processor. When the cache fetches the signature requested by the watchdog processor, the main processor is generating cache hits and executing one instruction per cycle.

It is interesting to note that the quicksort program, with the largest memory penalty, shows a roughly negligible performance overhead. This can be explained because performance penalty is related to how well the instruction cache is used by the test program, not by the static memory footprint of the program itself.

Another interesting conclusion comes from the analysis of the results for the matrix program. Being a small program consisting mainly in nested loops (that is, a good use of the instruction cache), shows the worst performance however. Analyzing the program source code, and from the simulation trace, it is evident that this program is dominated by the data memory access time. As the program performs lots of reads and writes, and there is no data cache, the probability of interference between a cache refill of the instruction cache due to a watchdog miss and the memory operation increases. If the cache refill has already started, the

Table 2: Performance penalty

Program	CPU cycles		Overhead
	Without watchdog	With watchdog	
Fft	892	940	5,38 %
Hanoi	2717	2732	0,55 %
Quicksort	510	520	1,96 %
Matrix	330	350	6,06 %
Queens	925	967	4,54 %

memory operation is delayed (as instruction and data paths share the AMBA bus) thus increasing the time the CPU needs to finish the program.

To our knowledge, there is no performance overhead analysis in the watchdog processors presented in the literature. However, taking the proposal in [3] as an example, the memory footprint increase sits between 12 % and 25 %. If the program being executed has few data movement instructions (like the hanoi towers test program), and due to the fact that signatures must be processed as normal instructions in that proposal, one could expect a performance overhead of 10 % in the best case.

3.3. Clock frequency and silicon overhead

In the case of field programmable devices, it is well known that different sub-elements can achieve very different operation frequencies. In our case, the main processor is the slowest element and the rest of the system (including watchdog, instruction cache and MMU) can use a clock with at least double frequency. That is, watchdog did not affect processor or cache clock.

We have taken advantage of this fact to reduce the complexity of the instruction cache, reducing the watchdog frequency to match main processor to simplify the design. Instead of creating a true dual-port cache, the cache has a single read port and some glue logic to provide read access for both processors at one half of its operating frequency. As the operating frequency of the processors is one half the cache frequency, both processors have read access at its maximum frequency and can be fed with data every cycle (on cache hits).

The same time multiplexing technique has been used in the MMU to deliver address translations for both processors in the same CPU cycle with a single access port MMU.

The whole system has been targeted to a Xilinx's Virtex2 device (a xc2v6000-4). To obtain the silicon overhead derived from the use of the watchdog processor, two versions have been created, synthesized and compared. The first one has a true single-port instruction cache and no watchdog and the second one incorporates all these elements previously described.

As the device technology incorporates many

different programmable elements (flip-flops, combinational logic functions, memory blocks, tristate buffers), there is no single value that can measure the silicon cost of both versions to perform a fair comparison. However, the Xilinx tools provide a total "equivalent" number of gates, a rough measure of system complexity that can be of some use. Comparing those numbers, the overall silicon overhead due to the inclusion of the error detection mechanism is 8,49 %.

4. Conclusions

The architecture of the HORUS processor has been presented. It is a RISC pipelined processor augmented with a concurrent error detection mechanism (a watchdog processor) specifically designed to minimize the CPU performance penalty. This goal is of primary importance to successfully employ fault tolerance mechanisms in the current market of field-programmable systems, where the clock frequency is far from the general-purpose personal computer ranges.

The performance overhead reduction is achieved modifying the standard meaning of branch and call instructions. This is only possible when the processor is modeled using a language like VHDL and thus architecture modifications are possible. The results obtained from several test programs show the performance penalty is 6 % or below, and that is negligible in some cases, depending on the program structure and its use of the instruction cache.

While memory requirements are fundamentally the same as previous proposals, performance benefits from the fact that signatures are neither fetched nor processed by the CPU. This fact and the results from the silicon studies (less than 8,5 % of silicon increase, the CPU clock is not affected) will promote the use of such a solution when a concurrent error detection mechanism is needed in a field-programmable device.

The HORUS processor has been designed using a synthesizeable subset of the VHDL language and it is currently under beta test. Characterization of the watchdog processor (percentage of errors detected, mean number of cycles to detect the error) is currently underway.

Bibliography

[1] Avresky, D., Grosspietsch, K.E., Jhonson, D.W., and Lombardi, F.: Embedded Fault Tolerant Systems. *IEEE Micro Magazine*, 8-11, Vol. 18, No. 5, 1998.

[2] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[3] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[4] Siewiorek, D.P.: Niche Sucesses to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 26-33, Pasadena, California, 1995.

[5] Rodríguez, F., Campelo, J.C, Serrano, J.J.: A Watchdog Processor Architecture with Minimal

Performance Overhead. *Proc. of the 21st Safety and Reliability Conference (SAFECOMP'02)*, Catania (Italy), Sept. 2002.

[6] Rodríguez, F., Campelo, J.C, Serrano, J.J.: A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream. *IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'2002)*, Vancouver, Canada, Nov. 2002.

A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream

F. Rodríguez, J. C. Campelo, and J. J. Serrano

Grupo de Sistemas Tolerantes a Fallos - Fault Tolerant Systems Group
Dept. Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, 46022 Valencia (Spain),
e-mail: {prodrig, jcampelo, jserrano}@disca.upv.es,
http://www.disca.upv.es/gstf

Abstract

Using a watchdog processor for concurrent error detection of a processor execution flow is a well-known technique to increase the dependability of a microprocessor system. Most approaches embed reference signatures for the watchdog processor into the processor instruction stream creating noticeable memory and performance overheads.

The *Interleaved Signature Instruction Stream* (ISIS) technique is a signature embedding technique that allows signatures to co-exist with the main instruction stream with a minimal impact on processor performance, without sacrificing error detection coverage or latency.

This technique has been implemented into HORUS, a MIPS-like RISC processor developed in VHDL. This paper presents the HORUS architecture novelties demanded by ISIS, discusses the performance impact of adding an ISIS watchdog processor and provides results of ISIS memory overhead. These results are compared against similar solutions previously presented in the literature.

1. Introduction

In the "Model for the Future" foreseen by Avizienis in [1] the urgent need to incorporate dependability to every day computing is clear: “Yet, it is alarming to observe that the explosive growth of complexity, speed, and performance of single-chip processors has not been paralleled by the inclusion of more on-chip error detection and recovery features”.

Efficient error detection is of fundamental

importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent *Error Detection Mechanism* (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experiments demonstrate [2, 3, 4, 5], a high percentage of non-overwritten errors results in control flow errors.

An application program is divided into branch-free intervals [5], called instruction blocks. A **derived signature** is a value assigned to each instruction block. The term *derived* means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using such opcodes to feed a Linear FeedBack Shift Register (LFSR). These values are calculated at compile time and used as reference by the EDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as *Embedded Signature Monitoring* (ESM). A **watchdog processor** [6] is a hardware EDM used to detect *Control Flow Errors* (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. In this case it performs signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their references. If any difference is found the error in the main processor instruction stream is detected and an *Error Recovery Mechanism* (ERM) is activated.

The *Interleaved Signature Instruction Stream* (ISIS) technique is an ESM technique that intersperses signatures and the main processor instructions, and allows the inclusion of a watchdog processor into a complex microprocessor system. It has been implemented into a complex MIPS-like RISC processor designed in VHDL called HORUS. To minimize the impact on the main processor performance of using signatures for concurrent error detection, the processor does not fetch or execute signatures.

The paper is structured as follows: The next section outlines ESM techniques previously proposed in the literature. Section 3 is devoted to introduce the HORUS architecture and how the watchdog processor is included into the system, to present the ISIS technique and the processor modifications in the section 4. Next section discusses how signatures impact main processor performance and a memory overhead comparison with similar work is performed afterwards, to finish with the conclusions.

2. Related work

Several hardware approaches using a watchdog processor and derived signatures for concurrent error detection have been previously proposed in the literature. The most relevant works are outlined below:

Ohlsson *et al.* present in [5] a watchdog processor built into a RISC processor. A

specialized `tst` instruction is inserted in the delay slot of every branch instruction, testing the signature of the preceding block. An instruction counter is also used to time-out an instruction sequence when a branch instruction is not executed in the specified range (signaling a branch deletion error). Other watchdog supporting instructions are added to the main processor instruction set to save and restore the value of the instruction counter on procedure calls.

The watchdog processor used by Galla *et al.* in [7] to verify correct execution of a communications controller of the Time-Triggered Architecture uses a similar approach. A `check` instruction is inserted in appropriate places to trigger the checking process with the reference signature that is stored in the subsequent word. In the case of a branch, the branch delay slot is used to insert an adjustment value for the signature to ensure the run-time signature is the same at the check instruction independent of the path followed. An instruction counter is also used by the watchdog. The counter is loaded during the check instruction and decremented for every instruction executed; a time-out is issued if the counter reaches zero before a new check instruction is executed. Due to the nature of the communications architecture, no interrupts are processed by the controller. Thus saving the run-time signature or instruction counter is not necessary.

The ERC32 is a SPARC processor augmented with parity bits and a program flow control mechanism presented by Gaisler in [8]. In the ERC32, a test instruction to verify the processor control flow is also inserted in the delay slot of every branch to verify the instruction bits of the preceding block. In his work, the test instruction is a slightly modified version of the original `nop` instruction and no other modifications to the instruction set is needed.

A different error detection mechanism is presented by Kim and Somani in [9]. The decoded signals for the pipeline control are checked in a per instruction basis and their references are retrieved from a watchdog private cache. If the run-time signature of a given instruction can't be checked because its reference counterpart is not found in the cache, it is stored in the cache and used as reference for future executions. No signatures or program modifications are needed because reference signatures are generated at run-time, thus creating no overhead. The drawback in this approach is that the watchdog processor can't check all instructions. An instruction can be checked only if it has been previously executed and its reference has not been displaced from the watchdog private cache to store other signatures. Although the error is detected before the instruction is committed and no overheads are created, the error coverage is poor as there is no guarantee a given instruction can be checked with a valid reference.

3. HORUS Architecture

The system (see Fig. 1) is built around a soft-core of a MIPS R3000 processor clone developed in synthesizeable VHDL [10] at

RTL level. We have called it HORUS[1].

HORUS is a 4-stage pipelined RISC processor running the MIPS-I and MIPS-II Instruction Set Architecture [11]. The internal bus for this SoC follows the AMBA's [12] multimaster Advanced High-Performance Bus (AHB) specification. Instructions and data are retrieved from external memory using two separate AHB masters to improve bus utilization.

This processor is provided with a Memory Management Unit (MMU) inside the System Control Coprocessor (*CP0* in the MIPS nomenclature) to perform virtual to physical address mapping, isolate memory areas of different processes and check correct alignment of memory references.

To minimize performance penalty, the instruction cache is designed with two read ports that can provide two instructions simultaneously, one for each processor. On a cache hit, no interference exists even if the other processor is waiting for a cache line refill because of a cache miss.

A single write port is provided to access the AHB bus, so it must be shared by both processors. When simultaneous cache misses happen, cache refills are served in a First-Come First-Served fashion. If they happen in the same clock cycle, the main processor is promoted.

This arrangement takes advantage of space locality in the application program to augment cache hits for signatures. As we use an ESM technique and signatures are interleaved with processor instructions, when both processors produce a cache miss they request the same memory block most of the times, as both reference words in the same program area.

No modification is needed in the processor instruction set due to the fact that signature instructions are neither fetched nor executed by the main processor. This allows us to maintain binary compatibility with existing software. If access to the source code is not possible, the program can be run without modification (and no concurrent flow error detection capability will be provided). This is possible because the watchdog processor and processor's modified architecture can be enabled and disabled under software control running with superuser privileges. If these features are disabled, our processor behaves as an off-the-shelf MIPS processor. Thus, if binary compatibility is needed for a given task, these features must be disabled by the OS every time the task resumes execution.

The watchdog processor is fed with the instructions from the main processor pipeline as they are retired. When these instructions enter the watchdog the run-time signatures and address parity bits are calculated at the same rate of the arrived instructions. When a block ends, these values are stored in a FIFO memory to decouple the signature checking process. This FIFO allows a large set of instructions to be retired from the pipeline while the watchdog is waiting for a cache refill in order to get a reference signature instruction. In a similar way, the FIFO can be emptied by the watchdog while the main processor pipeline

[1]HORUS is the name of an ancient Egyptian god, son of ISIS and OSIRIS

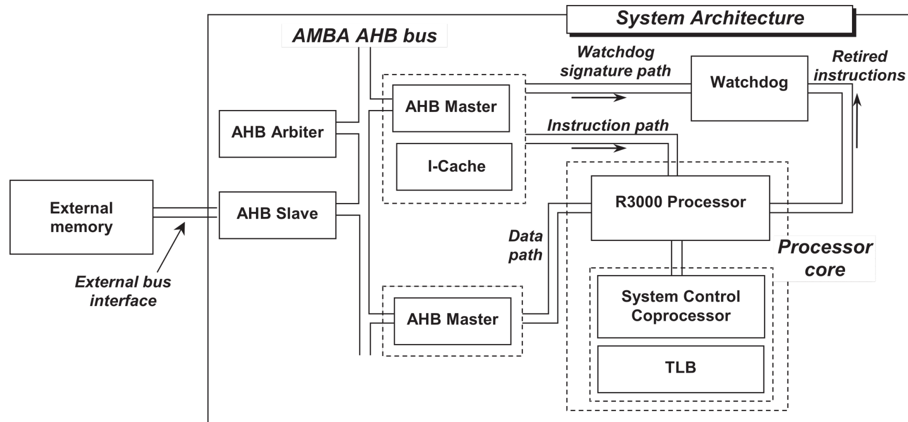

Figure 1: System architecture

is stalled due to a memory operation. When this memory is full, the pipeline if forced to wait for the watchdog checking process to read some data from the FIFO.

4. Interleaved Signature Instruction Stream

Contrary to other ESM techniques where signatures are placed in the delay slot of the branch instruction finishing the block, signatures are placed at the beginning of every basic block in the ISIS scheme [13]. These references incorporate, among other checking mechanisms, block signature and block length.

Besides error detection capabilities obtained from the block signature, and due to the fact that the block reference word includes the block length and that it can be retrieved by the watchdog processor as soon as the block begins, branch insertion and branch deletion errors can be detected. The signature instruction encoding has been designed in such a way that a main processor instruction can not be misinterpreted as a watchdog signature instruction. This provides an additional check when a branch instruction is executed by the main processor. This check consists in the requirement to find a signature instruction immediately preceding the first instruction of every block. This also helps to detect a CFE if a branch erroneously reaches a signature instruction, because the used encoding will force an illegal instruction exception to be raised.

Additional checking related with the signature instruction type and partial jump address verification are also included.

4.1. Processor Architecture Modifications

Signatures are used by the watchdog processor only and not processed in any way by the main processor. To achieve this iso-

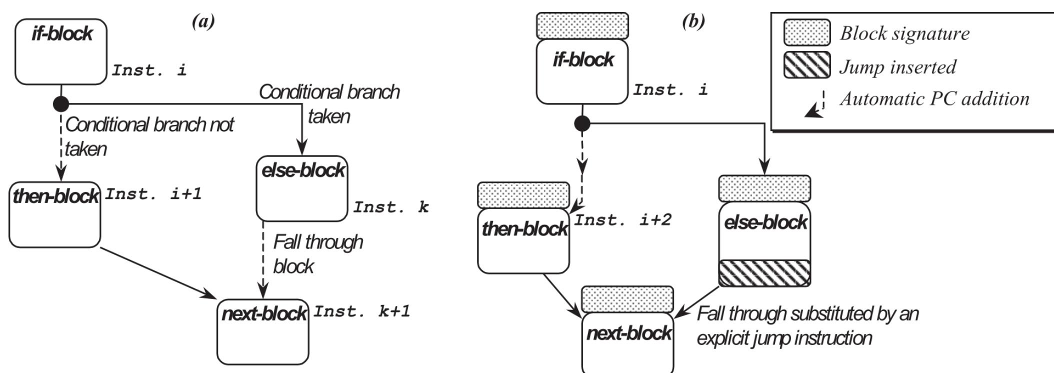

Figure 2: An `if-then-else` example (a). After block signatures and jump insertion (b)

lation from the main processor, they must be skipped at run-time or by means of jump instructions.

Isolating the reference signatures from the instructions fed into the processor pipeline results in a minimal performance overhead in the application program. Slight architecture modifications are needed in the main processor in order to achieve it.

First of all, when a conditional branch instruction ends a basic block, a second block follows immediately. The second block's signature sits between them, and the main processor must skip it. In order to effectively jumping over the signature, the signature size is added to the Program Counter if the branch is not taken.

In the same way, when a procedure call instruction ends a basic block the next one to be executed after the procedure returns immediately follows. Again, the second block's signature must be taken into account when calculating the procedure return address. And again, this is achieved by an automatic addition of the signature size to the PC.

Additions to the PC mentioned above can be automatically generated at run-time because the control unit decodes a branch or procedure call instruction at the end of the block. The instruction is a clear indication that the block end will arrive soon. As the processor has a pipelined architecture, the next instruction is executed in all cases (this is known as the *branch delay slot*), so the control unit has a clock cycle to prepare for the addition. Despite the fact that the instruction in the delay slot is placed after the branch, it logically belongs to the same block, as it is executed even if the branch is taken.

However, in the case of a *block fall-through*[13] the control unit has no clue to determine when the first block ends, so the signature can not be automatically jumped over. In this case, the compiler (currently, only the gnu C compiler, `gcc`) explicitly adds an unconditional jump to skip it. This is the only case where a processor instruction must be added in order to isolate main processor from the signature stream. Figure 2a shows an example of an `if-then-else` construct with a fall-through block that

Table 1: Memory overhead analysis

Program	Blocks	Instrs	Delay slots	Jumps	Signatures	Total overhead
fft	20	281	20	11	33	44 (15.66 %)
hanoi	21	118	21	0	23	23 (19.50 %)
quicksort	199	844	165	36	203	239 (28.32 %)
heapsort	52	315	45	9	56	65 (20.63 %)
matrix	25	139	20	5	27	32 (23.02 %)
queens	54	305	46	11	59	70 (22.95 %)

needs such an addition (shown in Fig. 2b).

The block length field used by the watchdog processor imposes restrictions to the length of the blocks, also. The compiler inserts jumps when the block length exceeds the maximum allowed, currently established in sixteen instructions.

5. Overhead analysis

Table 1 show some measurements obtained by a modified version of the `gcc` compiler. This has been tailored to produce block information about the compiled programs. The data produced include the number of sequential blocks and instructions of the original source, and how many of them are finished by a branch/call instruction and its associated delay slot. When the compiler is instructed to generate ISIS signatures, it also provides the number of inserted jumps (due to fall-through blocks or too large blocks) and signatures.

The jumps column includes the number of jumps inserted by the compiler. When a jump instruction is inserted, a nop instruction may be also inserted to fill its delay slot if no instruction of the original block is schedulable. In this case, the nop instruction is also counted in.

The analysed programs have been compiled with no optimization options, and so the results shown are worst case values. These programs are:

- Fft is the Fast Fourier Transform applied to a random set of values. It is a good representation of sequential programs, as it has a single block with 168 (!) instructions that ISIS must split into several blocks to accommodate them to its maximum block length.
- Hanoi, quicksort and heapsort solves the hanoi tower problem and sort a randomly initialized array of numbers, respectively.
- Matrix solves an integer matrix multiply and queens solves the classic nine queens placement problem

ISIS presents a memory overhead similar to those shown by similar approaches before,

as outlined in the next section. However, the additional error detection capabilities provided (block length, jump address and instruction signatures) compensates for this overhead, taking into account that the main processor has only to process the inserted jumps. This means a reduced impact on its execution performance contrary to previous techniques which demand CPU cycles to process signatures.

5.1. Comparison with related work

A purely software approach to concurrent error detection was evaluated by Wildner in [14]. This control flow EDM is called *Compiler-Assisted Self Checking of Structural Integrity* (CASC) and it is based on address hashing and signature justifying to protect the return address of every procedure. At the procedure entry, the return address of the procedure is extracted from the link register into a general-purpose register to be operated on. The first operation is the inversion of the LSB bit of the return address to provide a misalignment exception in the case of a CFE. An `add` instruction at each basic block is inserted to justify the procedure signature and, at the exit point, the final justifying and reinversion of the LSB bit is calculated and the result is transferred to the link register before returning from the procedure. In the case of a CFE, the return address obtained is likely to cause a misalignment exception thus catching the error. The experiments carried out on a RISC SPARC processor resulted in a memory codesize overhead for the SPECint92 benchmarks varying from 0 % to 28 % depending on the run-time library used and the benchmark itself.

The hardware watchdog of Ohlsson *et al.* presented in [5] use a `tst` instruction per basic block, taking advantage of the branch delay slot of a pipelined RISC processor called TRIP. One of the detection mechanisms used by the watchdog is an instruction counter to issue a time-out exception if a branch instruction is not executed during the specified interval. When a procedure is called two instructions are inserted to save the block instruction counter and another instrucion is inserted at the procedure end to restore it. Their watchdog code size overhead is evaluated to be between 13 % and 25 %.

6. Conclusion

We have outlined ISIS signature embedding technique and how it has been implemented into HORUS, a soft-core of a pipelined RISC processor.

The modifications demanded by signature embedding to the original processor architecture have been discussed. These modifications are very simple and can be enabled and disabled by software with superuser privileges to maintain binary compatibility with existing software. No specific features of the processor has been used, so the port of ISIS to a different processor architecture is quite straightforward.

Memory performance overhead has been

analyzed using a modified version of the `gcc` compiler to extract the program information needed. The resulting overhead (from 15 % up to 28 %) is comparable with other signature embedding methods previously proposed in the literature.

Several advantages distinguishes the ISIS technique, however. Error detection capabilities include signature processing, block length count, block type verification, and others.

We are currently obtaining experimental data on performance penalty and error detection coverage and latency. As expected, initial results show a small performance overhead between 0.5 % and 6 %, depending on the test program. These results are achieved because both processors share not only the instruction cache, but temporal and spatial locality also. That is, only a small subset of watchdog fetches interfere with the normal processor execution.

Acknowledgements

This work is supported by the Spanish Government *Comisión Interministerial de Ciencia y Tecnología* under project CICYT TAP99-0443-C05-02.

Bibliography

[1] Avizienis, A.: Building Dependable Systems: How to Keep Up with Complexity. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 4-14, Pasadena, California, 1995.

[2] Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. *Proc. of the 19th Fault Tolerant Computing Symposium (FTCS-19)*, 340-347, Chicago, Illinois, 1989.

[3] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[4] Gaisler, J.: Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-25)*, 42-46, Seattle, Washington, 1997.

[5] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[6] Mahmood, A., McCluskey, E.J.: Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Transactions on Computers*, 37(2): 160-174, 1988.

[7] Galla, T.M., Sprachmann, M., Steininger, A., Temple, C.: Control Flow Monitoring for a Time-Triggered Communication Controller. *Proceedings of the 10th European Workshop on*

Dependable Computing (EWDC-10), 43-48, Vienna, Austria, 1999.

[8] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[9] Kim, S., Somani, A.K.: On-Line Integrity Monitoring of Microprocessor Control Logic. *Proc. Intl. Conference on Computer Design: VLSI in Computers and Processors (ICCD-01)*, 314-319, Austin, Texas, 2001.

[10] *IEEE Std. 1076-1993: VHDL Language Reference Manual.* The Institute of Electrical and Electronics Engineers Inc., New York, 1995.

[11] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture.* MIPS Technologies, 2001.

[12] *AMBA Specification rev2.0.* ARM Limited, 1999.

[13] Rodríguez, F., Campelo, J.C., Serrano, J.J.: A Watchdog Processor Architecture with Minimal Performance Overhead. *To be presented at the SAFECOMP'2002*, Catania, Italy, September 2002.

[14] Wildner, U.: Experimental Evaluation of Assigned Signature Checking With Return Address Hashing on Different Platforms. *Proc. of the 6th Intl. Working Conference on Dependable Computing for Critical Applications*, 1-16, Grainau, Germany, 1997.

Improving the Interleaved Signature Instruction Stream Technique

F. Rodríguez, J. C. Campelo, J. J. Serrano

Dept. Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
C/ Camino de Vera S/N, 46022 - Valencia
SPAIN
{prodrig, jcampelo, jserrano}@disca.upv.es,
http://www.disca.upv.es/gstf

Abstract

Control flow monitoring using a watchdog processor is a well-known technique to increase the dependability of a microprocessor system. Most approaches embed reference signatures for the watchdog processor into the processor instruction stream creating noticeable memory and performance overheads. A novel embedding signatures technique called Interleaved Signatures Instruction Stream has been recently presented. Targeted to processors included into field-programmable devices, its main goal is to reduce the performance penalty induced by the watchdog processor in previous proposals. The work presented here improves the ISIS technique and offers a solution to the memory overhead without sacrificing performance, thus yielding a better overall architecture we have called OSIRIS: Another Interleaved Signature Instruction Stream.

Keywords *Error detection, Embedded signature monitoring, Reliability, Fault-tolerance, Microprocessors*

1. Introduction

In the "Model for the Future" foreseen by Avizienis in [1] the urgent need to incorporate dependability to every day computing is clear: "Yet, it is alarming to observe that the explosive growth of complexity, speed, and performance of single-chip processors has not been paralleled by the inclusion of more on-chip error detection and recovery features".

Efficient error detection is of fundamental

importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent Error Detection Mechanism (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experiments demonstrate [2, 3, 4, 5], a high percentage of non-overwritten errors results in control flow errors.

Siewiorek states in [6] that "To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users". A fault-tolerant technique can be considered transparent only if results in minimal performance overhead in silicon, memory size or processor speed.

Although redundant systems can achieve the best degree of fault-tolerance, the high overheads implied limit their applicability in every day computing elements.

The work presented here provides concurrent detection of control flow errors with minimal impact on the system performance, memory consumption and silicon sizes. No modifications are required into the architecture or the instruction set of the processor used as testbed in order to add a new instruction for the watchdog processor. The error detection capabilities can be enabled and disabled under software control to allow binary compatibility with existing software. The watchdog processor is very simple, and its design can be applied to other processors as well.

This work is derived from the Interleaved Signature Instruction Stream (ISIS) to improve its memory overhead.

The paper is structured as follows: The next section is devoted to introduce some basic terms in the field of watchdog processors and it is followed by the outline of the ISIS technique. Section 4 presents the OSIRIS technique and the resulting system architecture where the watchdog is embedded. Some discussion on performance and memory overhead with similar work is performed afterwards, to finish with the conclusions.

2. Introduction to watchdog processors

A minimal set of basic terms taken from [5] is needed to understand the overall system. A *branch-in* instruction is an instruction used as the target address of a branch or call instruction (for example, the first instruction of a procedure or function). A *branch-out* instruction is an instruction capable to break the sequential execution flow, conditionally or unconditionally (for example, a conditional branch or a procedure call instruction). A *basic block* is a sequence of instructions with no branch-in instructions except the very first one and no branch-out instructions except possibly the last one.

A *derived signature* is a value assigned to each instruction block. The term derived means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using such opcodes to feed a Linear Feed-

back Shift Register (LFSR). These values are calculated at compile time and used as reference by the EDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as *Embedded Signature Monitoring* (ESM). A *watchdog processor* is a hardware EDM used to detect *Control Flow Errors* (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. In this case it performs signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their references. If any difference is found the error in the main processor instruction stream is detected and an *Error Recovery Mechanism* (ERM) is activated.

3. The HORUS processor

In [7], a novel technique to embed signatures into the processor's instruction stream is presented. Its main goal is the reduction of the performance impact of the watchdog processor and it is targeted to processors included into field-programmable devices.

This technique, called ISIS (Interleaved Signature Instruction Stream), hash the watchdog processor signatures and application processor's instruction in the same memory area. Signatures are interleaved within instruction basic blocks, but these instructions are never fetched nor executed by the main processor.

3.1. System Architecture

The ISIS technique has been implemented in the HORUS processor [8], a soft-core clone of the MIPS R3000 [9] RISC processor (see Fig. 1). It is a four stage pipelined processor with a complete Memory Management Unit and instruction cache. The external memory and peripherals are accessed trough an AMBA AHB bus [10].

The original processor architecture has been augmented with a watchdog processor. The instruction cache is modified to include two read ports to provide simultaneous access to both processors (main and watchdog).

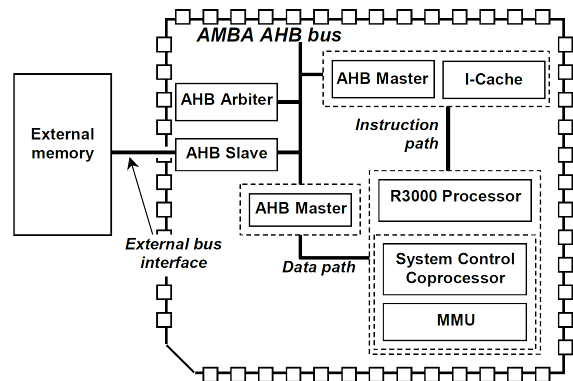

Figure 1: The initial HORUS architecture.

The watchdog processor (see Fig. 2) receives the main processor instructions as they are retired from the pipeline, performing the run-time calculations at the same rate the instructions are retired. When a basic block is finished, the run-time values are stored in a FIFO memory. The

checking process between reference signatures and run-time values reads from this FIFO and the instruction cache to perform the match.

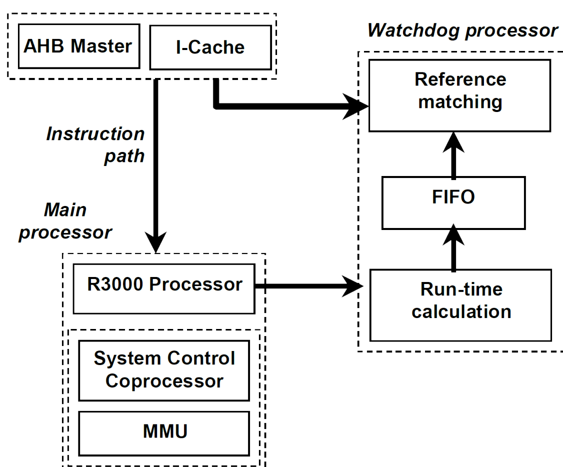

Figure 2: Modified architecture to include the watchdog processor using ISIS.

In Fig. 3, basic blocks for a conditional if-then-else statement are shown. Instruction addresses are shown in square brackets and lines indicate legal paths between blocks. Continuos lines are used to differentiate paths taken by means of an explicit jump from paths between blocks implicitly followed by the processor simply because instructions are executed in sequential order. These implicit paths are signaled with dashed lines.

For example, the if-block ends with a conditional branch instruction targeted at instruction j. In the case the condition is met, the branch is taken and the execution flow is explicitly changed to address j. In the case the condition is not met the branch is not taken and execution continues with the next instruction (at address $i + 1$). However, this sequential flow implicitly moved the execution flow from the if-block to the then-block.

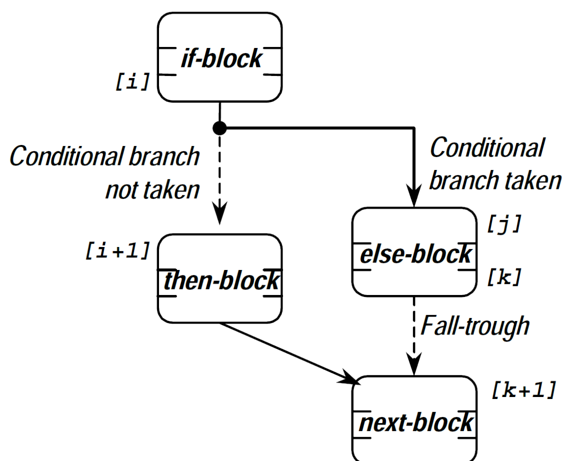

Figure 3: Basic blocks for a conditional if-then-else statement.

The main processor architecture has been modified to automatically skip watchdog signatures when possible. When the processor executes a conditional branch, the instruction that follows is automatically skipped even if the branch is not taken. The same applies when a procedure call is executed and the return address has to be stored.

These architectural modifications create word gaps in the main processor instruction stream immediately following branches and calls. And a specialized compiler to store watchdog signatures uses precisely these gaps.

In Fig. 4, the basic blocks of the same conditional statement are shown after the watchdog signatures have been added following the ISIS technique. In this figure, for example, the processor jumps over the

watchdog signature at address $i+1$ because the processor executes a conditional branch instruction at address i.

If the condition is met, the next instruction to be executed is stored at address j; if not, the processor skips the word after the branch and executes the instruction at address $i+1$.

Not all signatures can be jumped over however. Following with the same example that is shown in Fig. 4, the signature at address $k+2$ can not be automatically skipped. This is because the processor has no hint to determine the else-block is finishing when the instruction at address k is executed. In this case, the specialized compiler inserts an unconditional jump at address $k+1$ to skip the signature targeting to the instruction at address $k+2$. This jump is required because the signature instructions are encoded in such a way that the main processor would consider them as illegal instructions, raising the corresponding exception.

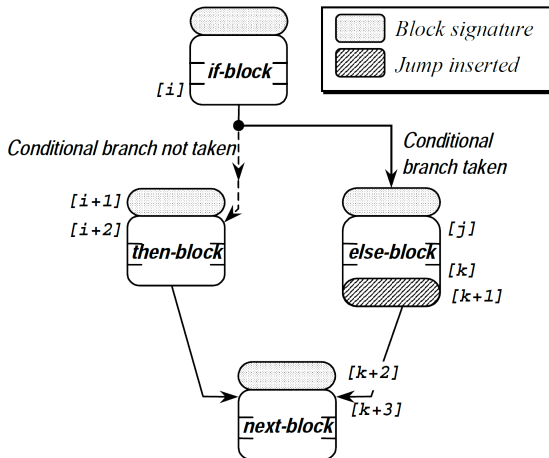

Figure 4: Basic blocks and interleaved signatures for the conditional statement.

3.2. Performance and memory overhead

In [11] some performance results from the HORUS processor are presented. These results are summarized in the table 1 and clearly show the goal of minimizing the performance penalty when the watchdog processor is in use has been achieved.

The test programs (fft, hanoi, quicksort and so on) are classic problems used as benchmarks as they reflect different program types: sequential, iterative and recursive. In all the cases, the penalty if 6 % or below, and strongly recursive programs will be affected by a negligible 0,5 % of performance loss.

Table 1: Performance results of the HORUS processor.

Program	CPU cycles		Overhead
	Without watchdog	With watchdog	
Fft	892	940	5,38 %
Hanoi	2717	2732	0,55 %
Quicksort	510	520	1,96 %
Matrix	330	350	6,06 %
Queens	925	967	4,54 %

Although performance overhead is minimal, memory consumption suffers from the fact that a signature word must be inserted for each basic block in the original program, and sometimes an explicit jump instruction must be also inserted to ensure the main processor will skip watchdog signatures.

Memory overhead results for the ISIS technique applied to the HORUS processor for the test programs above were presented in [8] and summarized in the table 2.

It can be easily observed that the resulting memory penalty is not the strong point of the ISIS technique, as the mean overhead is 24,14 % and can reach the 28 % for some test cases. Although quite large, these numbers are roughly the same results offered by other watchdog processors previously presented in the literature [5, 12].

4. Improving the Memory Overhead: OSIRIS

A new embedding technique is proposed here. It is a mixture between classical approaches like those in [5, 13] and ISIS. It is aimed to reduce memory overhead without sacrificing performance.

Classical watchdog processor approaches embed signatures in the delay slot of branch instructions. The delay slot is the instruction that immediately follows a branch, and in a RISC processor is always executed, even if the branch is taken. That is, it is stored immediately after the branch instruction but it logically belongs to the same basic block the branch belongs to. Modern compilers do instruction reordering to select an appropriate instruction just before the branch in order to fill the delay slot. If no instruction can be selected, the slot is filled with a no-operation instruction. Instead of performing instruction reordering, the delay slot is always filled with the block signature if a watchdog processor is used.

The memory overhead results of those solutions are similar to ISIS. They are however easier to implement, as signature checking is executed like any other arithmetic operation of the Execution Unit (EU). That is, the instruction used to trigger the signature checking process is like any other instruction.

This is the weakest point of those approaches. Signatures must be processed by the main processor like any instruction, so the performance loss is larger that the 6 % of ISIS.

To summarize, the watchdog processor should be a separate unit, not included into the processor execution unit to minimize performance loss. At the same time, creating a completely independent stream of signature instructions as proposed with the ISIS technique or using dedicated instructions like a classic approach result in a large memory overhead.

To solve memory overhead and separate the watchdog processor from the execution unit of the main processor we have developed another embedding technique. We have called Another Interleaved Signature Instruction Stream (OSIRIS) to this new technique.

The main idea behind this technique is to take advantage of unused fields of instructions of a basic block to store the signature of the block. Only when there is no enough room in these unused fields a new instruction is inserted. Even in this last case, the instruction is a no-operation instruction for the main processor, and its unused fields

Table 2: Memory consumption of the watchdog signatures in the HORUS processor.

Program	Original instr.	Inserted instr.	Signatures added	Overhead
Fft	281	11	33	15,66 %
Hanoi	118	0	23	19,50 %
Quicksort	844	36	203	28,32 %
Matrix	139	5	27	23,02 %
Queens	305	11	59	22,95 %

will store the remaining signature bits.

The MIPS instruction set, like most RISC processors instruction sets, is designed with the goal to achieve simplicity in the decoding process. This, combined with the fact that all instructions has the same length (32 bits), tend to create instructions with some unused fields.

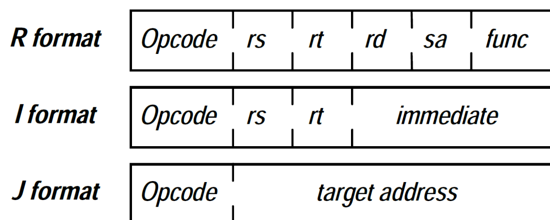

Figure 5: MIPS instruction formats.

The instruction formats of the HORUS processor is shown in Fig. 5, where rs, rt ad rd are register numbers. From the three instruction formats, the R format (the most used one) has between 5 and 15 unused bits, depending on the exact instruction.

This fact has been used in the MIPS16 architecture to create an instruction with the specific goal to reduce the length of every instruction from 32 to 16 bits. We propose to fill these unused bits with the block signature information.

Analyzing the instructions used by test programs mentioned in the previous section, the mean number of unused bits is 5,8 bits per instruction. And taking into account that the mean length of a basic block in a RISC processor sits between 7 and 8 instructions [14], the mean number of available bits in a block is more than enough to store its signature.

There are always small blocks that do not provide enough bits. In this case, a nop instruction is inserted at the end of the block. The unused bits of this instruction will provide enough room for the remaining bits of the block signature.

4.1. System Architecture using OSIRIS

The resulting system architecture is now simpler, as shown in Fig. 6. The instruction cache has now a single read port, and the main instructions enter both the main processor pipeline (to be executed) and the reference matching part of the watchdog processor (to extract the block signature).

The watchdog processor is more complex however, as it has to filter the main proces-

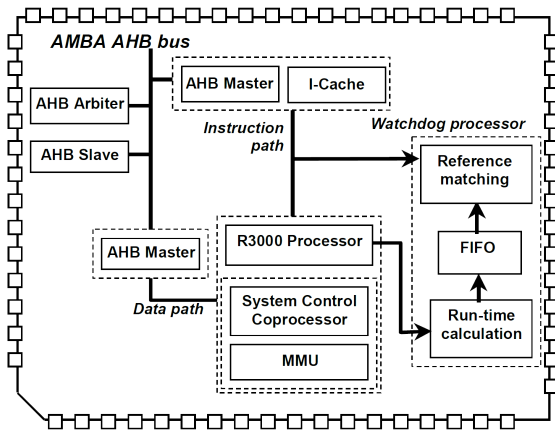

Figure 6: Processor architecture using the OSIRIS technique.

sor instructions to extract the block signature bits from unused fields of the executed instructions.

In order to maintain binary compatibility with existing software, the watchdog processor can be enabled and disabled under software control running with superuser privileges.

4.2. Performance and Memory Analysis

Although we have no experimental results yet to assess the performance or memory overhead of the OSIRIS technique, a preliminary analysis is possible.

As blocks larger than 6 instructions do not usually require an additional nop to store the block signature, there is no performance or memory penalty for those blocks.

For the other blocks, a single instruction is added, just like the proposal in [13], except that this last one requires this insertion for every block. As the ISIS technique requires a signature instruction per block, and a jump instruction for some blocks, its memory overhead must be larger than the OSIRIS technique here proposed.

With regard to performance, it is evident that performance with OSIRIS is better compared with classic approaches, as fewer instructions are inserted. In both cases, instructions added must be processed by the main processor and consume CPU cycles.

Comparing performance overhead of OSIRIS with ISIS results it is no so obvious. ISIS performance overhead is very small, and comes from the instruction cache accesses mainly. Only when performance results with the same test cases modified to embed signatures following the OSIRIS technique the comparison would be possible and fair.

5. Conclusion

We have presented a novel technique to embed signatures into the execution flow of a RISC processor that provides a set of error checking procedures to assess that the flow of executed instructions is correct.

All these checking mechanisms are performed in a per block basis, in order to reduce the error detection latency of our hardware Error Detection Mechanism.

We have called Another Interleaved Signature Instruction Stream (OSIRIS) to our signature embedding technique to reflect

this work is derived from the recently presented ISIS technique, whose goal is to minimize performance loss in processors included into field-programmable devices. The OSIRIS objective is to maintain performance and reducing the memory overhead, thus promoting the use of concurrent error detection mechanisms in new designs.

The main idea behind this technique is to take advantage of unused fields of instructions of a basic block to store its signature. Only when there is no enough room in these unused fields a new instruction is inserted. Even in this case, the instruction is a no-operation instruction for the main processor, and its unused fields will store the remaining signature bits.

No modifications are required to the processor architecture or its instruction set. In order to maintain binary compatibility with existing software, the watchdog processor can be enabled and disabled under software control running with superuser privileges.

The watchdog processor is very simple, and its design can be applied to other processors as well. No specific features of the processor have been used, so the port of OSIRIS to a different processor is quite straightforward. The memory overhead will vary depending on the number of unused bits in the processor instruction encoding.

Performance and memory overhead have been analyzed by comparison with other methods although we haven't performed a methodical study yet. As a few instructions are added to the original program, the performance is expected to remain basically unaltered.

Bibliography

[1] Avizienis, A.: Building Dependable Systems: How to Keep Up with Complexity. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 4-14, Pasadena, California, 1995.

[2] Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. *Proc. of the 19th Fault Tolerant Computing Symposium (FTCS-19)*, 340-347, Chicago, Illinois, 1989.

[3] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[4] Gaisler, J.: Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-27)*, 42-46, Seattle, Washington, 1997.

[5] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[6] Siewiorek, D.P.: Niche Sucesses to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 26-33, Pasadena, California, 1995.

[7] Rodríguez, F., Campelo, J.C, Serrano, J.J.: A Watchdog Processor Architecture with Minimal Performance Overhead. To be presented at the *21st Safety and Reliability Conference (SAFECOMP'02)*, Catania (Italy), 2002.

[8] Rodríguez, F., Campelo, J.C, Serrano, J.J.: The HORUS Processor. To be presented at the *XVII Conference on Design of Circuits and Integrated Systems (DCIS'2002)*, Santander, Spain, 2002.

[9] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture.* MIPS Technologies, 2001.

[10] *AMBA Specification rev2.0.* ARM Limited, 1999.

[11] Rodríguez, F., Campelo, J.C, Serrano, J.J.: Delivering Error Detection Capabilities into a Field Programmable Device: The HORUS Processor Case Study. Submitted to the *IEEE International Conference on Field-Programmable Technology (FPT'2002)*, Hong Kong, China, 2002.

[12] Wildner, U.: Experimental Evaluation of Assigned Signature Checking With Return Address Hashing on Different Platforms. *Proc. of the 6th Intl. Working Conference on Dependable Computing for Critical Applications*, 1-16, Grainau, Germany, 1997.

[13] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[14] Hennessy, J.L., Patterson, D.A.: *Computer Architecture. A Quantitative Approach*, 2nd edition, Morgan-Kauffmann Pub., Inc., 1996.

Improving the Interleaved Signature Instruction Stream Technique

F. Rodríguez, J. C. Campelo, J. J. Serrano

Dept. Informática de Sistemas y Computadores,
Universidad Politécnica de Valencia
C/ Camino de Vera S/N, 46022 - Valencia, Spain
prodrig@disca.upv.es, jcampelo@disca.upv.es, jserrano@disca.upv.es

Abstract

Control flow monitoring using a watchdog processor is a well-known technique to increase the dependability of a microprocessor system. Most approaches embed reference signatures for the watchdog processor into the processor instruction stream creating noticeable memory and performance overheads. A novel embedding signatures technique called Interleaved Signatures Instruction Stream has been recently presented. Targeted to processors included into field-programmable devices, its main goal is to reduce the performance penalty produced by the watchdog processor in previous proposals. The work presented here is an improvement of this technique and offers a solution to the memory overhead without sacrificing performance, thus yielding a better overall architecture. We have called this improved technique OSIRIS: Another Interleaved Signature Instruction Stream.

Keywords *Concurrent error detection; embedded signature monitoring; fault-tolerance*

1. Introduction

Efficient error detection is of fundamental importance in dependable computing systems. Although hardware redundancy can achieve the best degree of fault-tolerance, the high overheads implied limit their applicability in every day computing elements. And due to the fact that the vast majority of faults are transient, the use of a Concurrent Error Detection Mechanism (CEDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. As experiments demonstrate [1, 2, 3, 4], a high percentage of non-overwritten errors results in control flow errors.

The work presented here provides a CEDM

of control flow errors occurred into a general microprocessor with minimal impact on the system performance, memory consumption and silicon. No modifications of the instruction set architecture are required and the CEDM can be enabled and disabled under software control to allow a complete binary compatibility with existing software. The watchdog processor is very simple, and its design can be applied to other RISC processors as well.

This work is derived from the Interleaved Signature Instruction Stream (ISIS) as a requirement to improve its memory overhead, the only drawback compared with previous proposals.

The paper is structured as follows: The next section introduces some basic terms in the field of watchdog processors and it is followed by the outline of the ISIS embedding technique and its implementation into a MIPS processor. Section 4 presents the OSIRIS technique and the resulting system architecture. Some discussion on performance and memory overhead is performed afterwards, to finish with the conclusions.

2. Watchdog processors

A minimal set of basic terms taken from [3] is needed to understand the overall system. A branch-in instruction is an instruction used as the target address of a branch or call instruction (for example, the first instruction of a procedure or function). A branch-out instruction is an instruction capable to break the sequential execution flow, conditionally or unconditionally. A basic block is a sequence of instructions with no branch-in instructions except the very first one and no branch-out instructions except possibly the last one.

A derived signature is a value assigned to each instruction block. The term derived means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using such opcodes to feed a Linear Feedback Shift Register (LFSR). These values are calculated at compile time and used as reference by the CEDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as Embedded Signature Monitoring (ESM). A watchdog processor is a hardware CEDM used to detect Control Flow Errors (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. If this is the case, it performs run-time signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their stored references. If any difference is found the error in the main processor instruction stream is detected and an Error Recovery Mechanism (ERM) is activated.

3. System architecture with ISIS

In [5], a novel technique to embed signatures into the processor's instruction stream is presented. Its main goal is the reduction of the performance impact of the watchdog processor compared with previous. This technique, called ISIS (Interleaved Signature Instruction Stream), hashes the watchdog processor signatures and application processor's instruction in the same memory area. Signatures are interleaved within instruction basic blocks, but these instructions are never fetched nor executed by the main processor, as described below.

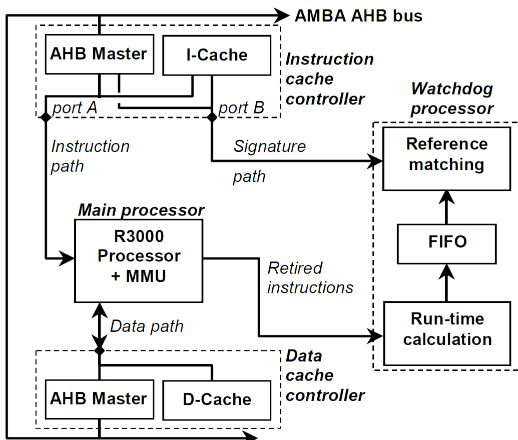

Figure 1: Modified system architecture to include the watchdog processor using ISIS.

The ISIS technique has been implemented in the HORUS processor [6], a soft-core clone of the MIPS R3000 [7] RISC processor (see Fig. 1). It is a four-stage pipelined processor with a complete Memory Management Unit (MMU) and instruction cache. The external memory and peripherals are accessed trough an AMBA AHB bus [8].

The MIPS original architecture is augmented with a watchdog processor. The instruction cache is modified to include two access ports, providing simultaneous access to both processors (application processor uses port A and watchdog uses port B).

The watchdog processor receives the main processor instructions as they are retired from the pipeline, performing at run-time the same signature calculations, and at the rate the instructions are retired. When a basic block is finished, the run-time values are stored in a FIFO memory. The checking process between reference signatures and run-time values reads from this FIFO and the instruction cache to perform the match.

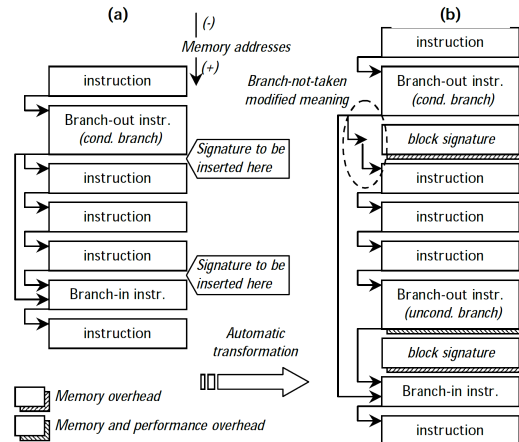

Figure 2: ISIS signature embedding example.

Signatures are interleaved with application instructions. The corresponding signature precedes every basic block of instructions. These signatures are automatically inserted into the program memory space by the com-

piler (see Fig. 2), a modified version of the GNU gcc compiler ported to the MIPS architecture. From the original application instructions (Fig 2.a), the compiler determines basic block signatures and placement, modifying the final executable to include them (Fig. 2b).

Conditional branch meaning when the condition is not met (that is, the actions taken by the processor when the branch is not taken) is modified such that the program flow jumps over the memory word after the branch. This is in contrast with the default meaning in a standard architecture: executing the instruction following the branch. This creates a gap between basic blocks when a conditional branch separates them. The compiler fills this gap with the block signature, which is not used by the main processor.

If a branch instruction does not separate the basic blocks, an unconditional branch instruction is required in order to create this gap, which is automatically inserted by the compiler also.

In [6] performance and memory overhead results from the HORUS processor using the ISIS technique are presented and summarized in the table 1. These results clearly show the goal of minimizing the performance penalty is achieved, but that memory consumption is not the strongest point of ISIS.

Table 1: Overhead results using ISIS.

Program	Memory overhead	Performance overhead
Fft	15,66 %	5,38 %
Hanoi	19,50 %	0,55 %
Quicksort	28,32 %	1,96 %
Matrix	23,02 %	6,06 %
Queens	22,95 %	4,54 %

In all cases, the performance penalty is below 6 % and strongly recursive programs will be affected by a negligible performance loss about 0,5 %. On the other hand, memory consumption suffers from the fact that a signature word must be inserted for each basic block in the original program, and sometimes an explicit jump instruction must be also inserted to ensure the main processor jumps over signatures. This makes the memory overhead reach the 28 % for some test cases. Although quite large, these numbers are roughly the same results offered by other watchdogs.

4. The OSIRIS approach

A new embedding technique is proposed here. It is a mixture between classical approaches like those in [3, 9] and ISIS, with the goal of reducing memory overhead without sacrificing performance.

Most ESM watchdog processors embed signatures in the delay slot of branch instructions. The delay slot is the instruction that immediately follows a branch, and is always executed in a RISC processor, even if the branch is taken. That is, it is stored immediately after the branch instruction but it logically belongs to the same basic block

the branch belongs to. Instead of performing instruction reordering, the delay slot is filled with the block signature if a watchdog processor is used.

The memory overhead results of those solutions are similar to ISIS. They are however easier to implement, as signature checking is executed like any other arithmetic operation of the Execution Unit (EU). That is, the instruction used to trigger the signature checking process is like any other instruction. And this is precisely their weakest point: as the main processor must execute instructions to check block signatures, performance is degraded.

To reduce memory overhead and at the same time separate the watchdog processor from the execution unit of the main processor we have developed a new embedding technique, called OSIRIS (Another Interleaved Signature Instruction Stream).

The main idea of this technique is to take advantage of unused fields of the application instructions to store the signature of the block. Only when there is no enough room in these unused fields a new instruction is inserted. If this is the case, the instruction must be a no-operation instruction for the main processor, and its unused fields will store the remaining signature bits.

The MIPS instruction set, like most RISC processors instruction sets, is designed with the goal of simplicity in the decoding process. This tend to generate instruction sets with some unused fields.

The instruction formats of the HORUS processor are shown in Fig. 3, where rs, rt ad rd are 5-bit register numbers. From the three instruction formats, the most used is the R format that has between 5 and 15 unused bits, depending on the exact instruction. This fact has been used in the MIPS16 architecture to create an instruction set with the specific goal to reduce the length of instructions from 32 to 16 bits. We propose to fill these unused bits with the block signature information.

R format	opcode	rs	rt	rd	sa	func
I format	opcode	rs	rt	immediate		
J format	opcode	address				

Figure 3: MIPS instruction formats.

Analyzing the instructions used by test programs mentioned in the previous section, the mean number of unused bits is 5,8 bits per instruction. And taking into account that the mean length of a basic block in a RISC processor sits between 7 and 8 instructions [10], the mean number of available bits in a block is more than enough to store its signature.

There are always small blocks that do not provide enough bits. In this case, a nop instruction is inserted at the end of the block. The unused bits of this instruction will provide enough room for the remaining bits of the block signature.

4.1. System Architecture using OSIRIS

The resulting system architecture is now simpler, as can be seen in Fig. 4 (only

modified elements are shown). The instruction cache has now a single access port, and the fetched instructions enter both the main processor pipeline (to be executed) and a new module of the watchdog processor (to extract the block signature).

The watchdog processor is more complex however, as it has to filter the main processor instructions to extract the block signature bits from unused fields of the executed instructions.

Although we have no experimental results yet to assess the performance or memory overhead of the OSIRIS technique, a preliminary analysis is possible.

As blocks larger than 6 instructions do not usually require an additional nop to store the block signature, there is no performance or memory penalty for them.

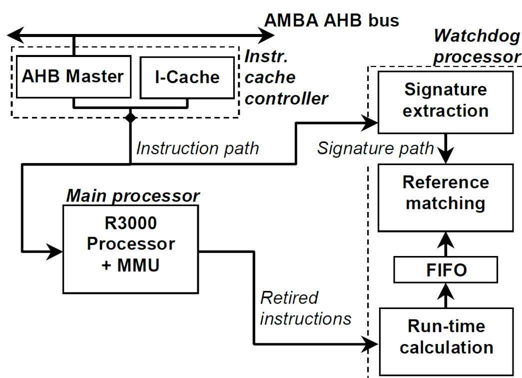

Figure 4: System architecture using OSIRIS.

For the other blocks, a single instruction is added, just like the proposal in [9], except that this last one requires this insertion for every block. As the ISIS technique requires a signature instruction per block, and a jump instruction for some blocks, its memory overhead must be larger than the OSIRIS technique here proposed.

With regard to performance, it is evident that performance with OSIRIS is better compared with previous approaches (apart from ISIS), as fewer instructions are inserted. Comparing performance overhead of OSIRIS and ISIS it is no so obvious. ISIS performance overhead is very small, and comes from the instruction cache simultaneous accesses mainly. It is the cache access pattern that dictates performance loss when the ISIS technique is in use, and from table 1 it is no direct relationship between memory and performance overheads. Only when performance results with the same test cases are available the comparison would be possible and fair.

5. Conclusion

We have presented a novel technique to embed signatures into the execution flow of a RISC processor that provides a set of error checking procedures to assess that the flow of executed instructions is correct. We have called it Another Interleaved Signature Instruction Stream (OSIRIS) to reflect this work is derived from the recently presented ISIS technique, whose goal is to minimize performance loss in processors included into field-programmable devices. The OSIRIS main point is to maintain performance and reduce memory overhead, thus promoting the use of CEDM in new designs.

We take advantage of unused fields of instructions of a basic block to store the

block's signature. Only if there is no enough room in those unused fields a new instruction is inserted. In order to maintain binary compatibility with existing software, the watchdog processor can be enabled and disabled under software.

The watchdog processor is very simple, and its design can be applied to other processors as well. No specific features of the processor have been used, so porting OSIRIS to a different processor is quite straightforward. The memory overhead will vary depending on the number of unused bits in the processor instruction encoding.

Performance and memory overhead have been analyzed by comparison with other methods although we haven't performed a methodical study yet.

Acknowledgements

This work is partially supported by the Spanish Government project CICYT TAP99-0443-C05-02 and the Valencian Community project CTIDIA/2002/27.

Bibliography

[1] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. *Proc. of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, 236-243, NewCastle Upon Tyne, U.K., 1990.

[2] Gaisler, J.: Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-27)*, 42-46, Seattle, Washington, 1997.

[3] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

[4] Wildner, U.: Experimental Evaluation of Assigned Signature Checking With Return Address Hashing on Different Platforms. *Proc. of the 6th Intl. Working Conference on Dependable Computing for Critical Applications*, 1-16, Grainau, Germany, 1997.

[5] Rodríguez, F., Campelo, J.C, Serrano, J.J.: A Watchdog Processor Architecture with Minimal Performance Overhead. *Proc. of the 21st Safety and Reliability Conference (SAFECOMP'02)*, Catania (Italy), Sept. 2002.

[6] Rodríguez, F., Campelo, J.C, Serrano, J.J.: Delivering Error Detection Capabilities into a Field Programmable Device: The HORUS Processor Case Study. *Proc. of the IEEE International Conference on Field-Programmable Technology (FPT'2002)*, pp. 418-422, Hong Kong, China, 2002.

[7] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture.* MIPS Technologies, 2001.

[8] *AMBA Specification rev2.0.* ARM Limited, 1999.

[9] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[10] Hennessy, J.L., Patterson, D.A.: *Computer Architecture. A Quantitative Approach*, 2nd edition, Morgan-Kauffmann Pub., Inc., 1996.

Control Flow Error Checking with ISIS

F. Rodríguez, J. J. Serrano

Grupo de Sistemas Tolerantes a Fallos - Fault Tolerant Systems Group,
Polytechnical University of Valencia, 46022, Valencia, Spain
{prodrig, jserrano}@disca.upv.es
http://www.disca.upv.es/gstf

Abstract

The Interleaved Signature Instruction Stream (ISIS) is a signature embedding technique that allows signatures to co-exist with the main processor instruction stream with a minimal impact on processor performance, without sacrificing error detection coverage or latency.

While ISIS incorporate some novel error detection mechanisms to assess the integrity of the program executed by the main processor, the limited number of bits available in the signature control word question if the detection mechanisms are effective detecting errors in the program execution flow. Increasing the signature size would negatively impact the memory requirements, so this option has been rejected. The effectiveness of such mechanisms is an issue that must be addressed. This paper details those checking mechanisms included within the ISIS technique that are responsible of the assessment of the integrity of the processor execution flow and the experiments carried out to characterize their coverage.

1. Introduction

With the advent of modern technologies in the field of programmable devices and enormous advances in the software tools used to model, simulate and translate into hardware almost any complex digital system, the capability to design a System-On-Chip (SoC) has become a reality even for small companies. With the widespread use of embedded systems in our everyday life, service availability and dependability concerns for these systems are increasingly important [1].

A SoC is usually modeled using a Hardware Description Language (HDL) like VHDL [2]. It allows a hierarchical description of the system and the designed elements interconnect much the same way as they would in a graphical design flow, but using an arbi-

trary abstraction level. It also provides IO facilities to easily incorporate test vectors, and language assertions to verify the correct behavior of the model during the simulation.

Efficient error detection is of fundamental importance in dependable computing systems. As the vast majority of faults are transient, the use of a concurrent *Error Detection Mechanism* (EDM) is of utmost interest as high coverage and low detection latency characteristics are needed to recover the system from the error. And as experiments demonstrate [3, 4, 5], a high percentage of non-overwritten errors results in control flow errors.

The possibility to modify the original architecture of a processor modeled using VHDL gives the SoC designer an unprecedented capability to incorporate EDM's which were previously available at large design companies only.

Siewiorek states in [6] that "To succeed in the commodity market, fault-tolerant techniques need to be sought which will be transparent to end users". A fault-tolerant technique can be considered transparent only if results in minimal performance overhead in silicon, memory size or processor speed. Although redundant systems can achieve the best degree of fault-tolerance, the high overheads imposed limit their applicability in everyday computing elements. The same limitation applies when a software only solution is used, due to performance losses. Siewiorek's statement can be also translated into the SoC world, to demand fault-tolerant techniques that minimize their impact on performance (the scarcest resource in such systems) if those techniques are to be used at all.

The work presented here is structured as follows: The next section is devoted to a minimal background on concurrent EDMs, specifically those using watchdog processors. A section of previous work follows, where the ISIS watchdog technique and its implementation into a SoC is described. The software support for this system is also outlined in this section.

Next section reports how the EDMs associated with the execution flow guarantee it; these are characterized, either theoretically or by means of some experiments. For those requiring experiments, the memory model is described in the corresponding subsection, along with the results obtained. The paper ends with the conclusions obtained and some further research opportunities.

2. Background

A minimal set of basic terms taken from [5] is needed to understand the overall system. A *branch-in* instruction is an instruction used as the target address of a branch or call instruction (for example, the first instruction of a procedure or function). A *branch-out* instruction is an instruction capable to break the sequential execution flow, conditionally or unconditionally (for example, a conditional branch or a procedure call instruction). A *basic block* is a sequence of instructions with no branch-in instructions except the very first one and no branch-out

instructions except possibly the last one.

A derived *signature* is a value assigned to each instruction block to be used as reference in the checking process at run-time. The term derived means the signature is not an arbitrarily assigned value but calculated from the block's instructions. Derived signatures are usually obtained xoring the instruction opcodes or using the opcodes to feed a Linear Feedback Shift Register (LFSR). These values are calculated at compile time and used as reference by the EDM to verify correctness of executed instructions.

If signatures are interspersed or hashed with the processor instructions the method is generally known as *Embedded Signature Monitoring* (ESM). A watchdog processor is a hardware EDM used to detect *Control Flow Errors* (CFE) and/or corruption of the instructions executed by the processor, usually employing derived signatures and an ESM technique. In this case it performs signature calculations from the instruction opcodes that are actually executed by the main processor, checking these run-time values against their references. If any difference is found the error in the main processor instruction stream is detected and an Error Recovery Mechanism (ERM) is activated.

The percentage of detected error is the error detection *coverage*, and the time from the error being active to the detection is the error detection *latency*. With both values any EDM can be characterized.

A *branch insertion* error is the error produced when the opcode of a non-branch instruction is corrupted and it is transformed into a branch instruction; from a watchdog processor perspective, this error is detected as a too early branch. A *branch deletion* error is the error produced when the opcode of a branch instruction gets corrupted and the instruction becomes a non-branch instruction; the watchdog detects this error condition as a too late branch.

Any error affecting a non-branch instruction other than branch insertion errors, do not affect the execution flow of the program and are not part of the structural integrity checking mechanisms.

3. Previous Work

In [7] a novel technique to embed signatures into the processor's instruction stream is presented. Its main goal is the reduction of the performance impact of the watchdog processor and it is targeted to processors included into embedded systems.

Using this technique, called ISIS (Interleaved Signature Instruction Stream), the watchdog processor signatures are hashed with the application processor's instructions in the same memory area. Signatures are interleaved within instruction basic blocks, but they are never fetched nor executed by the main processor.

Signature control words (or simply signatures) are placed at the beginning of every basic block in the ISIS scheme (see Fig. 1). These references incorporate, among other checking mechanisms, the op-

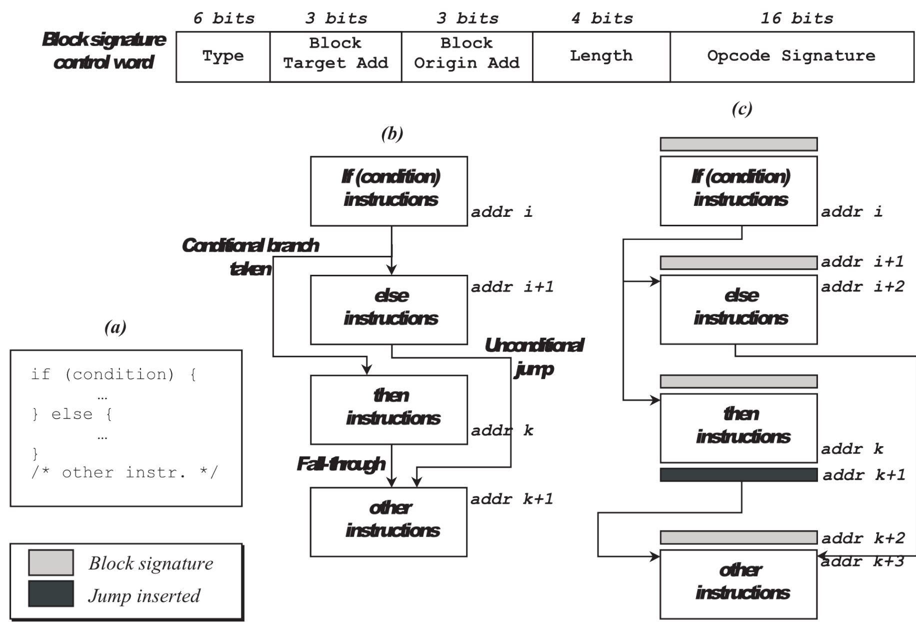

Figure 1: ISIS signature control word and signature insertion process: (a) high-level language snippet, (b) original blocks at assembly stage, and (c) after code is instrumented with signatures

code signature field: a polynomial CRC of the block instruction bits to detect the corruption of any instruction (non-CFE errors and branch insertion and deletion errors as well). Using a polynomial redundancy check 100 % of single bit errors and a large percentage of more complex error scenarios can be detected.

Besides error detection capabilities obtained from the opcode signature, and due to the fact that the block reference word includes the block length, branch insertion and branch deletion errors are detected.

The signature word encoding has been designed in such a way that a main processor instruction can not be misinterpreted as a watchdog signature instruction. This provides an additional check when the main processor executes a branch instruction. This check, called Branch Start, consists in the requirement to find a signature instruction immediately preceding the first instruction of every block. This also helps to detect a CFE if a branch erroneously targets a signature instruction, because the encoding will force an illegal instruction exception to be raised.

Under the assumption of single bit errors, the block length allows the watchdog processor to detect all branch insertion and branch deletion errors. Additional checking mechanisms related with the signature word instruction type and jump address guard bits are also included.

The Block Address is a check process that uses one of the address check fields in the

signature word (Block Origin Address or Block Target Address) to verify the correctness of the address of the target instruction when a branch is taken. The difference between the addresses of the branch instruction and the target instruction is computed at compile time, and a checksum is calculated and stored into the signature word. At run-time, when the processor breaks the execution sequence taking a branch, the actual addresses employed by the processor are used, inside the watchdog processor and following the same algorithm used by the compiler, to calculate another checksum. In the absence of errors, both must match; any mismatch will trigger the watchdog's error detection procedure.

This two EDMs, Block Start and Block Address, form the basic elements used by the watchdog processor to guarantee the integrity of the processor's execution flow. And the work presented in this paper shows their error coverage characteristics, using them separately and combined.

To reduce performance overhead the main CPU should not process signatures in any way. With this objective in mind, the CPU is designed to skip an instruction per basic block while maintaining the normal instruction sequencing. These architectural modifications create word gaps in the main processor instruction stream immediately following branches and calls. A specialized compiler uses these gaps to store watchdog signatures words.

With this arrangement, two completely independent interleaved instruction streams coexist in our system: the application instruction stream, which is divided into blocks and executed by the main processor and the signature stream, used by the watchdog processor.

Isolating the reference signatures from the instructions fed into the processor pipeline results in a minimal performance overhead in the application program. More information about this signature embedding technique can be found in [7].

The ISIS technique has been implemented in the HORUS processor [8], a soft-core clone of the MIPS R3000 [9] RISC processor (see Fig. 2). It is a four stage pipelined processor with a complete Memory Management Unit and instruction cache. The external memory and peripherals are accessed through an AMBA AHB bus [10]. This processor is provided with a Memory Management Unit (MMU) to perform virtual to physical address mapping, isolating memory areas of different processes and checking correct alignment of memory references. The watchdog processor is fed with the instructions from the main processor pipeline as they are retired.

The original processor architecture has been augmented with an ISIS watchdog processor. The instruction cache is modified to include two read ports to provide simultaneous access to both processors (main and watchdog processors).

The watchdog calculates run-time signatures at the same rate of the processor pipeline. When a block ends, these values are stored into a FIFO memory to decouple the checking process. This FIFO allows a large set of instructions to be retired from

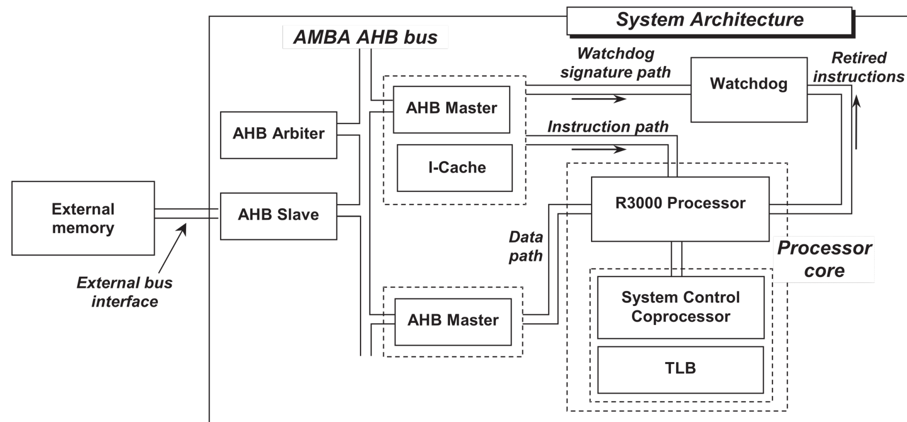

Figure 2: HORUS processor and overall system architecture

the pipeline while the watchdog is waiting for the block reference signature word. In a similar way, the watchdog can empty the FIFO while the main processor pipeline is stalled due to a memory operation. When this FIFO memory is full, the main processor is forced to wait for the watchdog checking process to read some data from it.

4. HORUS Compiler Support

The GNU *gcc* compiler already provides a port to target MIPS processors. As its source code is freely available it was the natural starting point to provide the required software support for the HORUS processor. The *gas* program (GNU Assembler) has the responsibility of the assembly stage in the compilation process, after program optimization passes and before the final linker stage.

The gas program and its supporting libraries have been modified to support the architecture of HORUS and its use of the ISIS technique via command line switches. As instructions are assembled,

1. If the current instruction is the target of a branch instruction, a new block starts and so its signature it is inserted.

2. If the current instruction is a branch, the next instruction will fill the branch delay slot and end the current block.

With this information and the opcode bits of the program instructions the assembler can calculate block signatures and insert them at appropriate places. No provisions are needed to modify the target of a branch or call instruction, as all instruction addresses are referenced using symbolic names (labels).

The software splits large sequences of instructions to accommodate the generated blocks to the length field of the signature control word. Reducing the number of in-

structions in a block increases the memory requirements, but it also reduces the latency from the error activation to its detection. While the length field would allow for blocks of up to 16 instructions, the actual block length could be smaller due to several reasons, most noticeably:

1. One of the instructions in the sequence is the target of a branch instruction. In this case, a signature must precede this instruction, so a new block must be created.
2. The use of variant frags. A variant frag is a combination of two different sequences of instructions generated by the assembler to solve the same task. For example, to store the address of a variable into a register, several sequences of instructions (and with different lengths) are possible using the MIPS instruction set, depending on the availability of a register pointer. If the symbol can not be resolved at assembly time, both sequences are generated. Obviously, only one of these would remain in the final executable, but the decision is delayed until the symbol address is resolvable. As the block size must be determined at the time of instruction generation, the approach taken has been conservative and the assumption that the larger sequence will remain is always followed. By the time the symbol is resolved, the blocks are already formed and their size can not be changed, so if the short sequence is finally selected the block will be shorter than 16 instructions.

5. Error Detection Coverage of CFEs

The Block Start EDM can be theoretically characterized, and its error coverage is 100 % as stated in proposition 1. The Block Address EDM requires some experiments to be carried out, as detailed below.

Proposition 1. *The Block Start checking mechanism ensures that all CFEs targeting an instruction other than the first instruction of a block are detected.*

Proof. A signature precedes the first instruction of a block. The watchdog processor uses the block initial address (being correct or not) as a memory reference to get the block's signature, retrieving it from the memory location immediately preceding this initial address. Given the fact that the bit patterns of signature words are selected not to match any instruction of the main processor, there are no instructions of the main processor that may be misinterpreted by the watchdog processor as a block signature.

So, in the case of a CFE targeting an instruction other than the first instruction of a block, the contents of the immediately preceding memory location is a processor instruction and not a signature word. Its bit pattern will not match any signature type in the watchdog processor, and the mismatch will trigger the error detection. □

Run-time calculation errors inside the main processor are not CFEs except if the incor-

rect value is an instruction address. Taking a branch or returning from a procedure, where a target instruction address must be calculated or retrieved from memory, are examples of such calculations. The opcode signature can not cover those calculations, as the original instruction is not corrupted.

Assessment of the effectiveness of the Block Address checking mechanism coupled with the Block Start check can only be performed by means of some kind of experimentation.

5.1. Experiment Setup

To determine the error detection coverage of EDMs applicable to CFEs a simulation model of the address calculation process has been created. This model mimics the performed operations of the actual processor at the execution of branches. Injecting faults into the model an erroneous target address is obtained and we are able to determine if the EDMs would detect it.

The simulation model consists of a large array of elements representing the processor's memory. Each element represents a block of sequential instructions with start address, length, signature, type of branch instruction, target address, etc. The type of branch instruction is important, as the address calculation process in the MIPS architecture is completely different if the instruction is a conditional branch or an unconditional jump. The former uses a program counter relative address and the later an absolute address.

Injecting a fault into the address calculation process in this model is as simple as randomly picking up the origin block, and simulating the effect of a single bit error at the branch.

Comparing the new, erroneous target address with the original one the fault masking probability is determined. A fault is masked if the calculation performed produces the same result as if there is no fault.

Using the erroneous address to compute the address guard bits and comparing them to those bits stored into the block signature word, the error detection probability of the Block Address EDM is obtained. The error detection probability of the Block Start EDM is obtained performing a search over the memory model to verify if the erroneous address matches the start of a block or not.

To simulate the effect of a single bit error in the address calculation process, a single bit of one of the operands or a single bit of the result is altered. Which value and which bit are chosen randomly. If it is an operand what is modified, the bit is changed before the target address is calculated. If it is the result, it is modified after the calculation is performed. Thus, a single bit error in the operands may propagate to adjacent bit positions to simulate the effect of a single or multiple bit error.

A synthetic workload is created filling the memory with blocks of random length, following a uniform distribution between 3 and 17 words. While the shortest block in the original MIPS architecture is 2 instructions long (the branch and the instruction at the branch delay slot), this block is augmented

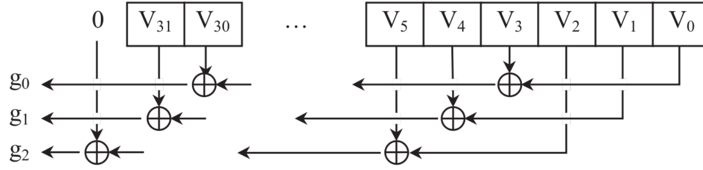

Figure 3: XOR tree to obtain checksum bits for a 3-bit address guard

with the block signature in HORUS (a signature has the same length of an instruction, it is a 32-bit word). The ISIS-modified gcc compiler limits the block length to accommodate it to the length field of the signature word, so no block larger than 17 words (16 instructions plus the signature) is allowed in our system. These length values match the mean length of sequential instructions, claimed to be between 7 and 8 [11].

Once the memory is filled, for each block the type of instruction at its end and the target block are chosen randomly. With this information, the address guard bits are calculated using the same algorithm internally used by the compiler and stored into the block structure for future reference.

This algorithm starts calculating the address difference between the branch and the target instructions. This 32-bit value is then compressed using a simple xor tree to obtain the address guard bits. Although the original proposal of ISIS reserves 3 bits for such guard, the xor tree is easily expandable to accommodate larger fields if space is available.

Figure 3 shows a representation of the xor tree for a guard fields of bits ($g_2g_1g_0$). Xoring alternating bits help the watchdog processor to detect multiple bit errors, where a single bit error into an operand propagates into a sequence of bit errors at the calculated result. Note that the 32-bit value calculated above ($V_{31..0}$) is padded with zeroes where necessary.

5.2. Results

Several fault injection campaigns have been carried out. Each campaign consists in the injection of 50,000 errors, and the experiments have been repeated a number of times with different random seeds to obtain their typical deviation, a statistical dispersion measurement.

Table 1: Block Start error coverage

Memory size	Mean (%)	Typ. deviation
64 Kbytes	45.14	0.287
256 Kbytes	50.44	0.339
1 Mbytes	56.53	0.133
2 Mbytes	59.21	0.212

To analyze the impact of the address guard field size, guards from 2 to 6 bits have been used in each experiment. The memory used by the application program has

Table 2: Block Address error coverage

Guard size	2 bits	3 bits	4 bits	5 bits	6 bits
Memory size	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev
64 Kbytes	96.52	98.31	98.70	99.29	99.37
	0.199	0.088	0.094	0.023	0.038
256 Kbytes	96.74	98.42	99.01	99.31	99.37
	0.055	0.042	0.042	0.018	0.028
1 Mbytes	96.81	98.48	99.15	99.36	99.40
	0.114	0.025	0.056	0.016	0.028
2 Mbytes	96.79	98.49	99.15	99.36	99.40
	0.060	0.054	0.020	0.040	0.034

been changed from 64Kbytes to 2Mbytes. A larger memory size theoretically increases the possibility of an erroneous branch to target the start of a block, and the error being undetected by the Block Start check.

Other elements incorporated into the HORUS processor incorporating checking mechanisms to detect CFEs but not explicitly included into the watchdog processor have not been included into our experiments as they do not characterize the error coverage we're trying to obtain from the inclusion of the watchdog. For example, the Memory Management Unit would trigger an exception if a branch targets a non-used memory area. Another check used by the main processor covering the same type of errors is the alignment check; all instructions fetched from memory must be aligned on a word boundary, or an exception is triggered. This means the results shown do not corresponds to the system error detection coverage, but only the coverage of the aforementioned EDMs.

The Table 1 summarizes the error coverage obtained with the Block Start mechanism alone, for each memory size.

As the results outline, the memory size has the inverse effect of what is theoretically expected. A larger memory increases, although moderately, the error coverage, despite the fact that there are more possibilities to target a block start erroneously. This can be explained by the fact that, at the same time, a larger memory means there are more possibilities the erroneous address fall inside the covered memory area.

The Table 2 shows the error coverage obtained with the Block Address mechanism for different address guard bits and memory sizes, and the combined error coverage is show in Table 3.

Another interesting result from the experiments carried out is the error length distribution, shown in Table 4. This table shows how a single bit error may propagate into a multiple bit error as the address calculation process takes place. Although data shown corresponds to one of the experiments only, the other experiments offer similar results and the data values have been omitted to

Table 3: Block Start and Block Address combined error coverage

Guard size	2 bits	3 bits	4 bits	5 bits	6 bits
Memory size	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev	Mean (%) Typ. dev
64 Kbytes	97.41	98.70	98.73	99.36	99.37
	0.139	0.084	0.092	0.023	0.038
256 Kbytes	97.97	98.68	99.31	99.34	99.37
	0.032	0.055	0.018	0.016	0.027
1 Mbytes	97.99	98.75	99.34	99.38	99.40
	0.070	0.045	0.044	0.013	0.027
2 Mbytes	98.55	99.27	99.35	99.38	99.94
	0.038	0.024	0.026	0.045	0.015

Table 4: Error length distribution

Error length	Mean (%)	Typ. deviation
0 (masked)	16.45	0.149
1	71.45	0.079
2	5.49	0.075
3	2.73	0.089
4	1.45	0.047
5	0.85	0.022
6	0.50	0.016
7	0.39	0.019
8	0.31	0.022
9	0.30	0.020
10	0.01	0.006

eliminate the redundancy. Error lengths above 10 bits have been also eliminated by its negligible impact.

As expected, the error length concentrates around single error bits, but percentages of masked errors, and multiple bit errors ranging from 2 to 4 bits are also noticeable.

6. Conclusions

The checking mechanisms to detect CFEs of the ISIS technique have been discussed, and its implementation on the HORUS processor has been outlined. This practical implementation has been complemented by a modified version of the ubiquitous C-language compiler gcc, to automatically insert signatures into the application program, lightening the programmer of most system reliability details.

Although the small number of bits reserved to check branch addresses could have generated some doubts about the effectiveness of the error detection mechanisms, this has been proven in contrary by the injection of faults into a model of the memory subsystem.

The model represents the contents of each block as a sequence of instructions preceded by the block's signature, and the address and length of each block is computed and stored for future reference. Single-bit errors have been injected into the model, and the

Block Start and Block Address EDMs have shown their effectiveness detecting CFEs.

Error coverage can be improved using an address guard field larger than the original 3-bit proposal. This requires reducing other checking fields, the opcode signature being the most promising alternative. Reducing this field could also reduce the error coverage of the associated mechanism (not described in this work) so the reduction requires further analysis.

Another interesting result depicted in this paper is the error length distribution in the address calculation process. Although single-bit errors are injected into the model, the arithmetic circuitry used in the address calculation process when a branch is taken helps the error to propagate as a multiple-bit error at the computed value. The error length distribution can be applied to other architectures using absolute or program counter relative addressing modes and would help future researchers to take into account this propagation when designing error detection mechanisms.

Acknowledgements

This work is supported by the Ministerio de Educación y Ciencia of the Spanish Government under project TIC2003-08106-C02-01.

Bibliography

[1] Avresky, D., Grosspietsch, K. E., Johnson, B. W., Lombardi, F.: Embedded fault tolerant systems. IEEE Micro Magazine, (1998) 18(5):8–11

[2] IEEE Std. 1076-1993: VHDL Language Reference Manual. The Institute of Electrical and Electronics Engineers Inc., New York (1995)

[3] Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In Proceedings of the 19th Fault Tolerant Computing Symposium (FTCS-19), Chicago, Illinois (1989) 340–347

[4] Czeck, E.W., Siewieorek, D.P.: Effects of Transient Gate-Level Faults on Program Behavior. In Proceedings of the 20th Fault Tolerant Computing Symposium (FTCS-20), NewCastle Upon Tyne, U.K. (1990) 236–243

[5] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. In Proceedings of the 22th Fault Tolerant Computing Symposium (FTCS-22), Boston, USA (1992) 316–325

[6] Siewiorek, D.P.: Niche Sucesses to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future. In Proceedings of the 25th Fault Tolerant Computing Symposium (FTCS-25), Pasadena, USA (1995) 26–33

[7] Rodríguez, F., Campelo, J.C., Serrano, J.J.: A Watchdog Processor Architecture with Minimal Performance Overhead. Lecture Notes in Computer Science (LNCS Series), Springer-Verlag ed. (2002) vol. 2434, 261–272

[8] Rodríguez, F., Campelo, J.C., Serrano, J.J.: The HORUS Processor. In Proceedings of the XVII Conference on Design of Circuits and Integrated Systems (DCIS 2002), Santander, Spain (2002) 517–522

[9] MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture. MIPS Technologies (2001)

[10] AMBA Specification rev2.0. ARM Limited (1999)

[11] Hennessy, J.L., Patterson, D.A.: Computer Architecture. A Quantitative Approach (2nd edition). Morgan-Kauffmann Publisher (1996)

Reducing the vhdl-based fault injection simulation time in a distributed environment

F. Rodríguez, J. C. Campelo, J. J. Serrano

Grupo de Sistemas Tolerantes a Fallos (Fault Tolerant Systems Group)
Dept. Informática de Sistemas y Computadores (DISCA)
Universidad Politécnica de Valencia, 46022-Valencia (Spain)
email: {prodrig, jcampelo, jserrano}@disca.upv.es

Abstract

In this paper we present a distributed simulation toolkit specially developed to help the researcher in the dependability assessment studies where the use of fault injection techniques into complex VHDL models are involved. Two mechanisms, restarting the simulator and restoring the model state, are evaluated. The tool architecture and results from experiments carried out on a complex SystemOnChip are presented to demonstrate its usefulness. These results clearly show that the selection of the proper mechanism results in a noticeable reduction of the simulation time when using our tool compared with a general-purpose workload distribution application.

1. Introduction

With the advent of modern technologies in the field of programmable devices and enormous advances in the software tools used to model, simulate and translate into real hardware almost any digital system, the capability to design a whole System-On-Chip (SoC) has become a reality even for small companies. With the widespread use of embedded systems in our everyday life, service availability and dependability concerns for these systems are increasingly important [1].

A SoC is usually modelled using a Hardware Description Language (HDL) like VHDL. It allows a hierarchical description of the system and the designed elements interconnect much the same way as they would in a graphical design flow, but using an arbitrary abstraction level. It also provides IO facilities to easily incorporate test vectors, and language assertions to verify the correct

behaviour of the model during the simulation.

Every *Error Detection Mechanism* (EDM) incorporated into the SoC to increase the system dependability must be characterised. This characterisation includes the probability to detect errors (*coverage*) and the time from fault activation to error detection (*latency*). *Fault injection* (FI) is a consolidated technique [2, 3] to assess mechanism's error detection properties, and it is also used to determine how errors propagate through the system, revealing which are the critical elements in the designed system.

Fault injection means a deliberated insertion of faults into a system in order to analyse its behaviour in the presence of errors and is defined in [3] as: "The dependability validation technique that is implemented by means of controlled tests where the observation of the behaviour of the system in presence of faults is explicitly induced by the deliberate introduction (injection) of faults in the system". Differences between fault injection and other experimental techniques are due to FI involves the whole system, both its physical component (hardware) and its behavioural component (software).

Fault injection may be performed during the design phase using a simulation system model (simulation-based fault injection) or during the prototype phase injecting faults in a system prototype or in the final system. When a simulation-based fault injection is used, faults must be added to the system model. To be useful, these faults should simulate the effect of real faults on the real system.

In error propagation studies, the trace of the injected model is compared against a fault-free simulation trace called the *golden run* in order to show if the fault has activated itself generating an error and the error propagation path. In EDM's characterisation studies, the trace from the injected simulation must also include enough information from the EDM itself to determine if the error has been properly detected or not.

Several simulation-based fault injection techniques have been proposed in the literature [4, 5, 6, 7, 8, 9] that use a HDL to describe the system and the faults to be injected. The use of simulator commands is an injection technique based on the use of commands to force the value of some signals in the VHDL model, thus generating a fault. As the fault is injected into the model at simulation time, the original model needs no modification or recompilation, making this technique a popular solution. This is the approach used by the tool presented here.

In order to achieve an adequate confidence level in the dependability results, the statistical analysis demands a large set of simulations (several thousands) to be carried out, even after pruning techniques [10, 11] are applied. In this simulation set, called an *experiment campaign*, we must decide what kind of faults must be considered, where to inject them, and when during the simulation run. Every run is the simulation of the model in the presence of a single fault.

This paper is structured as follows. In the next section, the motivation for this work

and its objective is presented followed by the description of the developed tool. Then, the SoC used for our experiment campaigns is briefly described, and the measures taken for different environments are presented. After this, the results of such measures are analysed, finishing with the work conclusions.

2. Motivation

The massive simulation workload for the fault injection campaign naturally fits in a distributed environment. Simulation runs are independent of each other, so they can be managed as different simulator executions. This is the approach used by general-purpose tools to achieve automatic resource sharing and load balancing on this kind of complex, heterogeneous environments [12, 13]. They help the SoC designer to carry out the simulations in a distributed environment and collect the result files, but they do not cover the campaign data generation or the analysis of the simulation results. For these tasks, a specialised tool is needed [4, 5, 6, 8, 14] easily coupled with a distributed simulation environment (if a general-purpose tool for the distributed environment is used).

The general-purpose tools use a workload model that translates every user task (a simulation run in our case) into a set of batch program executions. This makes no use of the capabilities of current available simulators [15], loosing a speed-up opportunity. If the speed-up lost is sufficiently large, it can justify the use of a specialised simulation management tool.

With a powerful simulator, a *restart* command exists to shift the simulated model time to zero, allowing several simulations to be carried out without the overhead of finishing the simulator program and executing it again. It is also possible the use of a *restore* command, shifting the model to a previously saved state, simulating from the restored time on. This restoring mechanism may be used to trim fault injection simulations, as the model behaves exactly as fault-free until the time the fault is injected.

Our research group has already developed a fault-injection tool, called VFIT [14]. It is powerful and mature fault injection tool for VHDL models. It includes a rich set of features in the field of fault injection, but lacks the distributed simulation capabilities mentioned above, so we must resort to a general-purpose tool for the simulation workload.

The software toolkit presented here fulfils all these questions, incorporating a simulation framework with full use of VHDL simulator commands to speed-up the simulation process. We have called this set of software elements the FIASCO toolkit (Fault Injection Aid Software COmponents). Among the features you can find in this toolkit are:

- Automated processes to generate the experiment campaign data. The user selects the signals to be traced in the simulation from a hierarchical view of the SoC model. A graphic interface is also used to specify the fault injection parameters (number and type of faults

to be injected, the distribution functions of the injection start time and length) and simulation options (simulation time, use of restart or restore commands, etc.).

- Use of a heterogeneous set of simulation hosts to distribute the simulation runs. FIASCO makes full use of the simulator commands to speed up the simulations. To avoid system overheads the simulator program is executed once per host. As the model does not change, a very fast restart command may be issued for every simulation run.

- Automatic collection of the result files and analysis, obtaining the statistical data the user has previously requested. A specific language has been developed to let the user express relationships from the golden run and the injected simulation in order to generate the experiment's dependability formulae.

The goal of the FIASCO toolkit and the experiments presented here is to determine if noticeable performance improvements are possible with the use of a specialised simulation framework, before including this feature in the next release of VFIT. As the work presented here is based on an interpreted language, it is expected that the compiled nature of VFIT will even increase the performance obtained with FIASCO.

3. The FIASCO toolkit

Modelsim is a very popular digital simulator from a mayor EDA Company that runs on a large variety of system architectures [15]. The graphic interface is built around an embedded Tcl/TK interpreter [16] giving the user an unlimited expandability with the use of tcl scripts that can be dynamically loaded and executed. These tcl scripts are text files that can add new commands or modify existing ones. The user can even create his own graphic interface using the TK toolkit. These capabilities are exploited by FIASCO to graphically assist the user in his dependability research.

This commercial simulator provides the commands mentioned in the previous section to speed up a set of simulations from a single program execution. These commands can be entered from a text console or from a batch script, and will be processed by the modelsim internal tcl interpreter. The simulator includes text-only batch simulations executing commands from a text script, and this is the approach used in our simulation hosts.

The FIASCO toolkit is built around two tcl scripts that integrate into modelsim once loaded and communicate themselves with TCP/IP sockets using a client/server protocol (see Fig. 1). The client (called FIASCO-C) generates the campaign data, performs the statistical analysis and controls the simulation in the distributed environment. The server (FIASCO-S) executes within the modelsim simulator in each simulation host. It receives simulation re-

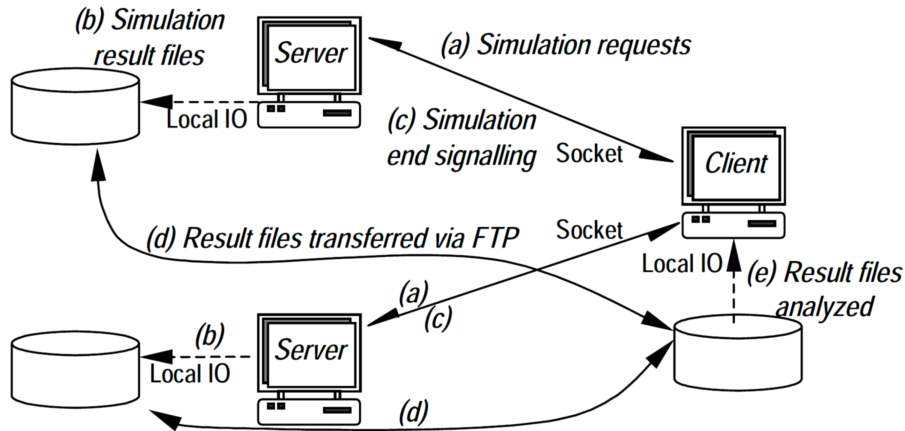

Figure 1: FIASCO toolkit component interconnection.

quests from the client and uses the simulator commands to carry them out. As FIASCO-S is an integral part of the simulator, it can be used on any system architecture supported by the simulator itself. The FIASCO-C script incorporates the experience gained and the technologies developed within the development of our group's tool VFIT mentioned before.

Sockets are used as the communication channel between the client and the simulation servers. To supports network failures, the connection between client and server uses an asymmetric protocol that is stateless in the server side. Steps followed to carry out a simulation campaign are as follows. First, the client (step a) assigns a set of simulations to every idle server. The server simulates and locally stores the result files (step b), signalling the client (step c) when the simulations have finished. The client reacts assigning a new set of simulations for the idle server, and collecting the result files (step d) from the simulation host using the ftp protocol. Once all the result files are available, the client analyses them (step e) to obtain the dependability statistics.

If the network fails for some reason, the connection socket between server and client closes, but the server continues simulating. If it can not signal the end of requested simulations to the client, it simply waits for the client to reconnect. When the client connects with a simulation server it first requests the server status to determine if the server is idle or not, so the protocol on the client side can resynchronise accordingly.

To ensure interoperability between client and servers, result files are plain text files. However, text-based trace files tend to be very large due to the file format the simulator uses (more than 40 Mbytes in our largest experiments – see next section for a description). To avoid running out of disk and wasting time in the ftp transfers, we have developed a new propietary format. This format is still human readable text, but it achieves a reduction ratio between 19 and 26 (depending on the trace itself). The fi-

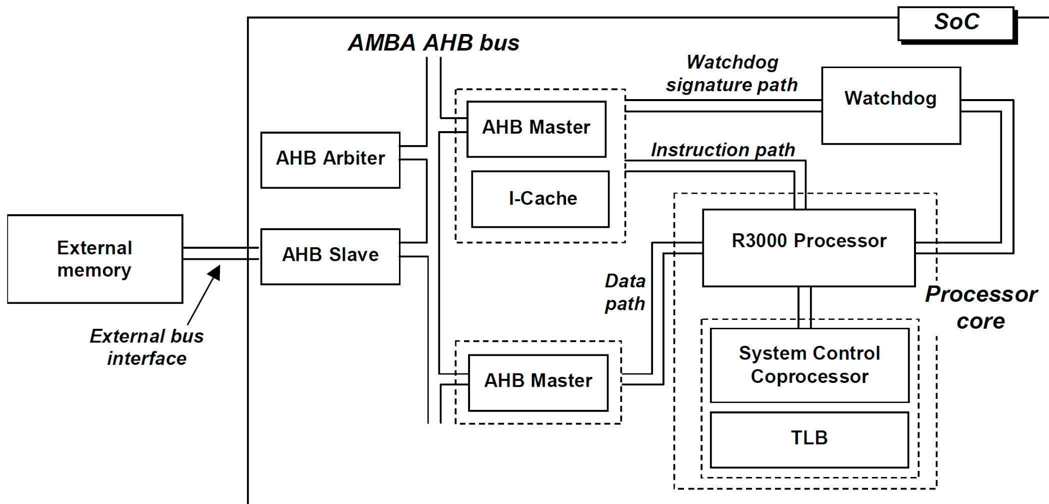

Figure 2: SoC architecture under test.

nal file is then compressed using a standard utility before the simulation is considered finished.

3.1. Simulation speed-up techniques

The FIASCO-S component is executed only once per host, and the simulator program is kept alive from simulation to simulation. This is radically different from a general-purpose approach, which would execute the simulator for every injection, with the corresponding OS overhead to load and unload the simulator program.

The user has the flexibility to decide that only the restart command is used. In this case, all simulations are carried out from time zero. The OS overhead to load and unload the program is substituted by the time the simulator needs to initialise the model state.

If the user decides to use the restore command, a number of state files (checkpoints) are generated before the first injection. When a simulation is started, the closest state is selected trimming the simulation time. However, it must be taken into account that a restore is much heavier than a simple restart, as the model state must be retrieved from a disk file. There also exists the initial overhead to create the checkpoints that must be also accounted.

4. Test description

To test the FIASCO toolkit we have used a complex SoC (see Fig. 2). This comprises a MIPS R3000 processor [17], an instruction cache and a set of AMBA bus elements [18] to connect the SoC with the external memories. We use a watchdog processor similar to the ones described in [19, 20] as our EDM using the Embedded Signature Monitoring

Table 1: No checkpoints experiments - total and mean simulation times (in seconds).

Total simulation time

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
1	22,61	77,11	235,55	475,68
10	179,44	733,64	2308,26	4699,60
50	875,57	3649,85	11597,22	23631,73
100	1742,93	7272,51	23204,49	47729,94

Mean simulation time

Simulated clock cycles			
100	*500*	*1500*	*3000*
22,61	77,11	235,55	475,68
17,944	73,364	230,826	469,96
17,5114	72,997	231,9444	472,6346
17,4293	72,7251	232,0449	477,2994

Table 2: 10 checkpoints experiments - total and mean simulation times (in seconds).

Total simulation time

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	212,68	683,97	2030,9	4073,23
50	850,51	3197,35	10098,24	20304,64
100	1647,03	6295,28	19743,52	40510,87

Mean simulation time

Simulated clock cycles			
100	*500*	*1500*	*3000*
21,268	68,397	203,09	407,323
17,0102	63,947	201,9648	406,0928
16,4703	62,9528	197,4352	405,1087

technique [20].

All the elements mentioned above have been developed in our group as synthesizable RTL models. We have also added a trace facility inside the processor's model, and a VHDL testbench to incorporate the external memories into the model. The program space is filled at start-up time with the Eratosthenes sieve prime number generator program inside an infinite loop. This allows us to arbitrarily change the number of simulated CPU clock cycles.

We have carried out a simple experiment in order to estimate our tool's performance improvement, studying the simulation time for a single simulation server for different testbench configurations. Both server and client execute onto the same machine, a PC box (a 1,1 GHz Athlon processor with 512 Mbytes of DDR-SDRAM) using Linux Mandrake. This arrangement eliminates the transfer of the result files from server to client, restricting experiment times to simulation

We vary the number of injections to be carried out between 10 to 100 to evaluate the number of injections a single server should perform in a medium to large machine pool. To study the simulation time for different complexity levels, we simply change the number of clock cycles to be simulated from 100 to 3000. These simulation cycles translate in simulation times from 22 seconds to 8 minutes for a single injection.

If we call T to the mean time to execute the simulator program for a single injection and n to the number of injections, a lower bound for the simulation time using a general-purpose tool is simply derived as $n \times T$. To evaluate the usefulness of using the restart and restore simulator commands, three different types of simulations

Table 3: 50 checkpoints experiments - total and mean simulation times (in seconds).

Total simulation time

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	425,13	895,02	2236,12	4253,16
50	1065,82	3380,29	10138,02	20325,9
100	1836,89	6421,62	20026,1	40209,05

Mean simulation time

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	42,513	89,502	223,612	425,316
50	21,3164	67,6058	202,7604	406,518
100	18,3689	64,2162	200,261	402,0905

Table 4: Simulation times (normalised).

No checkpoints

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	0,79	0,95	0,98	0,99
50	0,77	0,95	0,98	0,99
100	0,77	0,94	0,99	1,00

10 checkpoints

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	0,94	0,89	0,86	0,86
50	0,75	0,83	0,86	0,85
100	0,73	0,82	0,84	0,85

50 checkpoints

	Simulated clock cycles			
n	*100*	*500*	*1500*	*3000*
10	1,88	1,16	0,95	0,89
50	0,94	0,88	0,86	0,85
100	0,81	0,85	0,85	0,85

are performed: using the restart command only (no checkpointing) and using 10 and 50 checkpoints to restore.

5. Experimental results

The measured times for the simulations using no checkpoints are shown in the Table 1. The total time is the actual measurement and the mean simulation time is derived from this value and the number of injections. Tables 2 and 3 show the same measurements for the experiments using checkpoints.

We normalise the mean simulation times using T as the simulation unit to compare performances of FIASCO and a general-purpose tool. The value of T is the time for a single simulation (row $n = 1$ in Table 1 above). Normalised values are shown in Table 4.

From Table 4 is evident the benefit from using a specialised simulation tool like FIASCO. The performance improvement grows up to 23 % using no checkpoints for short simulations and up to 15 % using checkpoints for long simulations. Interestingly, increasing the number of checkpoints does not produce a noticeable performance increment. Although simulated time is shorter as the number of checkpoints increase, because checkpoints are closer to the time the fault must be injected, this does not translate in an overall improvement. This may be due to the increase in the initial overhead to generate more checkpoints.

The checkpoint generation overhead can be even counterproductive when injections demand a short time to simulate. If injections can be simulated fast, the simulation time saved is negligible and does not compensate the large time required to generate the checkpoints. These results do not agree

with the ones presented in [11]. The authors argue that checkpointing is always beneficial, and that it is only necessary to trade-off the optimum number of checkpoints. Those results are however obtained from a much simpler model (an 8-bit microcontroller with no cache memories or MMU), so the checkpoints need much less time to be generated and restored and the generation overhead can be easily compensated. The complexity differences in the models used may explain this discrepancy.

6. Conclusions

In this paper, a specialised distribution simulation framework for dependability assessment using fault injection into VHDL models has been presented. The performance improvement obtained with our FIASCO toolkit using a simple experiment has been also presented and the results are very interesting.

FIASCO is especially well suited for fault injection experiments needing a large number of short simulations, as it is the case for dependability assessment of low-latency error detection mechanisms. The performance benefit ranges from 23 % to 27 % per simulation, depending on the total number of simulations a single host must perform and the speed-up technique used. These figures prove the usefulness of such a tool in the dependability assessment field.

Not all simulation cases benefit so much, however. Performance improvements drop to 15 % for large simulations when a single host must carry out a large number of simulations and is a moderate 10 % for a large enterprise pool where every host has a few simulations to solve.

Acknowledgements

This work is partially supported by the Spanish Government's Comisión Interministerial de Ciencia y Tecnología under the project reference CICYT TAP99-0443-C05-02.

Bibliography

[1] Avresky, D., Grosspietsch, K.E., Jhonson, D.W., and Lombardi, F.: Embedded Fault Tolerant Systems. *IEEE Micro Magazine*, 8-11, Vol. 18, No. 5, 1998.

[2] Laprie, J., C.: Dependable computing: concepts, limits and challenges. *Proc. of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, 42-54, Pasadena, California, 1995.

[3] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J. C., Laprie, J. C., Martins, E., Powell D.: Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 166-182, vol. 16, 1990.

[4] Boue, J., Petillon, P., Crouzet, Y.: MEFISTO-L: a VHDL based fault injection tool for the experimental assessment of fault tolerance. *Proc. of the 28th Fault Tolerant Computing Symposium (FTCS-28)*, pp. 168-73, 1998.

[5] Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., Karlsson, J.: Fault Injection into VHDL Models: The MEFISTO Tool. *Proc. of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, pp. 66-75, 1994.

[6] Gil, D., Baraza, J. C., Busquets, J. V., Gil, P. J.: Fault Injection with simulation in VHDL and application to a simple microcomputer system. *Proc. of the 5th International Conference on Advanced Computing*, pp. 466-474, 1997.

[7] Sieh, V., Tschäche, O., Balbach, F.: VHDL-based Fault Injection with VERIFY. TR-5/96, University of Friedrich-Alexander, Computer Architecture Department, Erlangen-Nuremberg, 1996.

[8] Sieh, V., Tschäche, O., Balbach, F.: VERIFY: Evaluation of Reliability Using VHDL models with Embedded Fault Descriptions. *Proc. of the 27th Fault Tolerant Computing Symposium (FTCS-27)*, pp. 32-36, 1997.

[9] Gil, D., Baraza, J. C., Busquets, J. V., Gil, P. J.: Fault Injection into VHDL models: Analysis of the Error Syndrome of a Microcomputer System. *Proc. of the 24th Euromicro Conference*, pp. 418-424, 1998.

[10] Berrojo, L., González, I., Corno, F., Sonza, M., Entrena, L., Lopez, C.: New Techniques for Speeding-up Fault-Injection Campaigns. *Proc. of the Design, Automation & Test in Europe (DATE 2002)*, pp. 847-852, Paris 2002.

[11] Parrotta, B., Rebaudengo, M., Sonza, M., Violante, M.: Speeding-up Fault-Injection Campaigns in VHDL models. *Proc. of the 19th Intl. Conference on Computer Safety, Reliability & Security (SAFECOMP 2000)*, pp. 27-36, Rotterdam 2000.

[12] Basney, J., Livny, M.: Deploying a High Throughput Computing Cluster. *High Performance Cluster Computing*, vol. 1, R. Buyya (Editor), Prentice Hall 1999.

[13] Zhou, S., Wang, J., Zheng, X., Delisle, P.: Utopia: A load sharing facility for large, heterogeneous distributed computing systems. University of Toronto, Computer Systems Research Institute, Toronto 1992.

[14] Baraza, J. C., Gracia, J., Gil, D., Gil, P. J.: A Prototype of a VHDL-Based Fault Injection Tool. Description and Application. *Journal of Systems Architecture*, special issue on Defect and Fault Tolerance in VLSI Systems, to appear.

[15] *Modelsim SE v5.5b Command Reference*. Model Technology Inc., 2001.

[16] Welch, B.: *Practical Programming in Tcl/TK, 3rd edition*. Prentice Hall, 2001.

[17] *MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture*. MIPS Technologies, 2001.

[18] *AMBA Specification rev2.0*. ARM Limited, 1999.

[19] Gaisler, J.: Concurrent Error-Detection and Modular Fault-Tolerance in an 32-bit Processing Core for Embedded Space Flight Applications. *Proc. of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, 128-130, Austin, Texas, 1994.

[20] Ohlsson, J., Rimén, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. *Proc. of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, 316-325, Boston, Massachusetts, 1992.

A Distributed Simulation Environment for Fault Injection Analysis on SoC Models

F. Rodríguez, J. C. Campelo, J. J. Serrano

Grupo de Sistemas Tolerantes a Fallos (*Fault Tolerant Systems Group*)
Departamento de Informática de Sistemas y Computadores (DISCA)
Universidad Politécnica de Valencia, 46022-Valencia (Spain)
email: {prodrig, jcampelo, jserrano}@disca.upv.es

Abstract

In this paper, we present a distributed simulation environment specially developed to help the researcher in dependability assessment studies that involve the use of fault injection techniques into complex VHDL models. The tool architecture and some results from experiments carried out on a complex SystemOnChip are presented. These results show a noticeable reduction of the simulation time when using our tool compared with a general-purpose workload distribution application.

1. Introduction and Motivation

With the advent of modern technologies in the field of programmable devices and enormous advances in the software tools used to model, simulate and translate into real hardware almost any complex digital system, the capability to design a whole System-On-Chip (SoC) has become a reality even for small companies. With the widespread use of embedded systems in our everyday life, service availability and dependability concerns for these systems are increasingly important [1].

Fault injection (FI) is a technique that is being consolidated in the Fault Forecasting area. It is defined in [2] as: "The dependability validation technique that is implemented by means of controlled tests where the observation of the behaviour of the system in presence of faults is explicitly induced by the deliberate introduction (injection) of faults in the system". When a simulation-based fault injection is used, faults must be added to the system model. The results of the injected model are compared against a fault-free simulation run called the *golden run*.

Several simulation-based fault injection techniques using a HDL language to describe the system under test have been proposed in the literature. One of these techniques uses simulator commands to force the value of the signals that connect the elements of the design, thus generating a fault. As the fault is injected into the model at simulation time, the original model needs no modification or recompilation, making this technique a popular solution. This technique is known as *forcing* and it used for the tool presented here.

To gather the statistical information needed to assess the model dependability results a large set of simulations (several thousands usually) must be carried out. Every simulation run of this campaign is the simulation of the model in the presence of a single fault.

The massive simulation workload for the fault injection campaign naturally fits in a distributed environment. Simulation runs are independent of each other, so they can be treated as different simulator executions. This independence allows the use of general-purpose resource sharing tools to manage simulations in complex heterogeneous distributed environments.

This approach makes no use of the features of current simulators, where a *restart* of the simulation model is possible, allowing several simulations to be carried out without the OS overhead of finishing the simulator program and executing it again. Those general-purpose tools lose a simulation speed-up opportunity here, and if it is sufficiently large, it can justify the use of a specialised simulation management tool.

A specialised tool is needed to generate the experiment campaign data anyway, with enough flexibility to let the user express the analysis that must be carried out. It must include the management of the simulations itself or be easily coupled with a general-purpose distributed simulation environment if such a system is used.

Our research group has already developed such tool, called VFIT [3]. VFIT is a powerful and mature fault injection tool for VHDL models but lacks the distributed simulation capabilities mentioned above, so we must resort to a general-purpose tool for the simulation workload distribution.

The software toolkit presented here fulfils these simulation questions, incorporating a simulation framework that uses simulator commands to inject faults into the system model and to speed-up simulations. We have called this set of software elements the FIASCO toolkit (Fault Injection Aid Software COmponents).

The goal of the FIASCO toolkit and the experiments presented here is to determine if noticeable performance improvements are possible with the use of a specialised simulation framework, before including this feature in the next release of VFIT. As the work presented here is based on an interpreted language, it is expected that the compiled nature of VFIT will even increase the performance obtained with FIASCO.

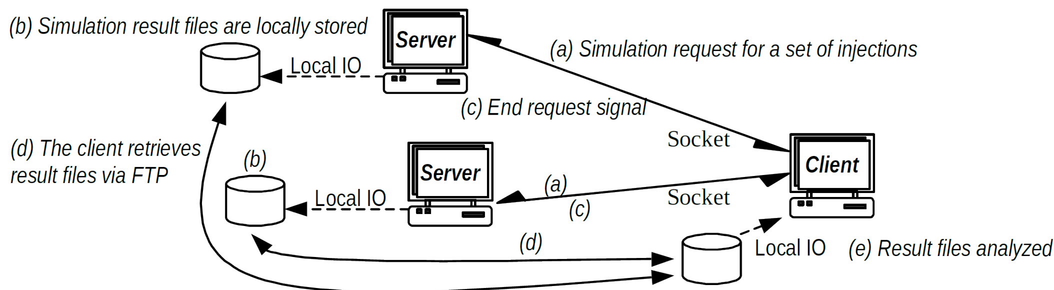

Figure 1: FIASCO toolkit component interconnection.

2. The FIASCO toolkit

Modelsim [4] is a very popular HDL simulator from a mayor EDA Company that runs on a large variety of system architectures. A built-in Tcl/TK interpreter is embedded into the simulator, giving the user an unlimited expandability with the use of scripts that can be dynamically loaded and executed. These scripts can add new commands or modify existing ones. This simulator provides a restart command to speed up a set of simulations from a single program execution. These capabilities are exploited by FIASCO to graphically assist the user in his dependability research and to manage simulations in a distributed environment.

The FIASCO toolkit consists of two scripts (see Fig. 1). The client (called FIASCO-C) generates the campaign data, performs the statistical analysis and controls the simulation in the distributed environment. The server (FIASCO-S) executes within the modelsim simulator in each simulation host. It receives simulation requests from the client and uses the simulator commands to carry them out. The FIASCO-C script incorporates the experience gained and the technologies developed within the development of our group's tool VFIT.

Sockets are used as the communication channel between the client and the simulation servers. To support network failures, the client/server protocol is used to issue requests and signal when those requests have been accomplished only. A simulation request is a request to inject a configurable number of faults into the system model using the restart command. The result of each simulation is locally stored into the server hard disk. Results files are retrieved by the client using the standard ftp protocol when the network is functioning.

If the network fails for some reason, the connection socket between server and client closes, but the server continues simulating until all the requested faults have been injected. If it can not signal the end of the request to the client, it simply waits for the client to reconnect. When the client connects with a simulation server it first requests the server status to determine if the server is idle or not, so the protocol on the client side can resynchronise accordingly. If idle, it issues a new simulation request in a

short message and opens an ftp connection to retrieve result files.

With this arrangement, intermittent network failures impact is minimised as the client sends simulation requests (very short messages, minimal network utilisation) even if result files can not be transferred.

To ensure interoperability between client and servers of different architectures, result files are plain text files. However, as text-based trace files tend to be very large due to the file format the simulator uses, we have developed a new format. This is still plain text, but it achieves a reduction ratio between 19 and 26 (depending on the trace itself). The final file is then compressed, for a total compression ratio above 99 % from the original text files.

3. Test description and experimental results

To test the FIASCO toolkit we have used a complex SoC system. It comprises a MIPS R3000 RISC processor, an instruction cache and a set of interconnection elements for the internal bus that connect the SoC with the external memories.

We have added a trace facility inside the processor's model, and a VHDL testbench to incorporate the external memories into the model. The program space is filled at start-up time with the Eratosthenes sieve prime number generator program inside an infinite loop. This allows us to arbitrarily change the number of simulated system clock cycles.

We have carried out a simple experiment in order to estimate our tool's performance, studying the simulation time for a single simulation server for different testbench configurations. This server is a SuSE Linux 6.4 in a medium range PC box and it is connected to the client through a 100Mbps LAN. To study the simulation time for different trace file sizes, we simply change the number of clock cycles to be simulated.

We will call T_C to the mean time to execute the simulator program for a single injection of C clock cycles, and n to the number of injections performed. With these definitions, a lower bound for the simulation time using a general-purpose tool is simply derived as $n \times T_C$.

This bound does not take into account the transfer time for the result files but it is included in the measured times from the FIASCO toolkit. However, as the final file to be transferred (after compression) is relatively small, the measured transfer time is negligible.

The study ranges n from 1 to 50 and sets C to 100, 500 and 1000 clock cycles. Results use $n \times T_C$ as a reference to obtain the relative percentage of performance improvement.

Simulation times are shown in the Table 1. The total time is the actual measurement and the mean simulation time is derived from this value and the number of injections. The relative improvement is the ratio of the total simulation time and the reference used for a general-purpose tool, $n \times T_C$.

Table 1: Total and mean simulation times (in seconds), relative performance improvement.

	Total simulation time			Mean simulation time			Relative improvement		
	Simulated clock cycles			*Simulated clock cycles*			*Simulated clock cycles*		
n	*100*	*500*	*1000*	*100*	*500*	*1000*	*100*	*500*	*1000*
1	58,35	101,49	179,69	58,35	101,49	179,69	0,00 %	0,00 %	0,00 %
2	75,29	180,45	337,94	37,64	90,23	168,97	35,48 %	11,10 %	5,97 %
5	141,32	419,00	808,49	28,26	83,80	161,70	51,56 %	17,43 %	10,02 %
10	244,91	825,30	1615,13	24,49	82,53	161,51	79,01 %	18,68 %	10,12 %
50	1156,73	4033,00	7984,95	23,13	80,66	159,70	86,78 %	20,53 %	11,13 %

Results clearly show the benefits from using a specialised simulation tool like FIASCO to manage simulations. The improvement grows with the number of simulations (n) to be carried out in the server host and it is inversely proportional to the time needed for a single stand-alone simulation (T_C). This means those small to medium size installations get more benefit per host (as a single host has more simulations to do). Results also show that the benefit increases for small to medium models (as larger models increase T_C).

4. Conclusions

In this paper, a specialised simulation framework for dependability assessment using fault injection into VHDL models has been presented. The performance improvement obtained with our FIASCO toolkit using a simple experiment has been also presented and the results are very interesting.

FIASCO is especially well suited for fault injection experiments needing a large number of short simulations. The performance benefit ranges from 35 % to 86 % per simulation, depending on the total number of simulations a single host must perform. These figures prove the usefulness of such a tool in the dependability assessment field.

The simulation techniques used in this work are no applicable to all simulation cases however. There are simulation experiments in the field of fault injection into VHDL were the use of the tool delivers a moderate improvement. In our current development, we are trying to surpass the percentage improvement for long simulations. This would make FIASCO a useful tool for a wider range of simulation studies.

Acknowledgements

This work is partially supported by the Spanish Government's *Comisión Interministerial de Ciencia y Tecnología* under project CICYT TAP99-0443-C05-02.

Bibliography

[1] Avresky, D., Grosspietsch, K.E., Jhonson, D.W., and Lombardi, F.: Embedded Fault Tol-

erant Systems. *IEEE Micro Magazine*, 8-11, Vol. 18, No. 5, 1998.

[2] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J. C., Laprie, J. C., Martins, E., Powell D.: Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 166-182, vol. 16, 1990.

[3] Baraza, J. C., Gracia, J., Gil, D., Gil, P. J.: A Prototype of a VHDL-Based Fault Injection Tool. Description and Application. *Journal of Systems Architecture*, special issue on Defect and Fault Tolerance in VLSI Systems, to appear.

[4] *Modelsim SE v5.5b Command Reference.* Model Technology Inc., 2001.

Parte III:

Conclusiones

Capítulo 13

Conclusiones

All generalizations are false, including this one.
Mark Twain

En este capítulo se resumen las aportaciones del presente trabajo, las diferentes publicaciones a las que ha dado lugar, y por último se plantean algunas de las líneas de investigación que quedan abiertas y que pueden guiar el trabajo futuro.

13.1. Introducción

Complementado con mecanismos de enmascaramiento o detección de errores en los datos, el uso de un procesador de guardia es una alternativa viable y de menor complejidad que la pura redundancia espacial necesaria en los sistemas duales.

La detección de errores de control de flujo mediante el uso de un procesador de guardia permite obtener un alto nivel de confianza en que el procesador ejecuta las instrucciones que se le requieran y en el orden en el que se le requieran, lo que abre la puerta a mecanismos de detección de errores de datos basados en software, lo que indudablemente abunda en un importante ahorro de costes.

Sin embargo, para que un mecanismo de este tipo sea utilizado en la práctica,

deben tenerse en cuenta una serie de condiciones:

- La arquitectura del procesador que se va a monitorizar debe ser modificada lo menos posible, a fin de facilitar la inclusión del procesador de guardia como un elemento añadido de seguridad sin requerir un complejo rediseño del procesador o de su juego de instrucciones.
- La utilización del procesador de guardia debe ser lo más transparente posible al usuario final, el programador. Si el programador debe disponer de conocimientos altamente específicos sobre los mecanismos de detección de errores incorporados en el sistema para poder utilizarlos de forma eficaz, podemos concluir que éstos serán descartados, infrautilizados o incluso peor, utilizados de forma incorrecta y generando una falsa confianza en el funcionamiento del sistema en presencia de errores.
- Cualquier mecanismo de detección de errores incorporado a un sistema que originalmente no disponía de él incurre en costes adicionales, sea en el diseño del sistema, en la complejidad del circuito resultante, en la memoria necesaria para su inclusión, o en las prestaciones obtenidas. En este último caso, la pérdida de prestaciones puede venir de dos fuentes diferentes, a saber: i) que el procesador necesite ciclos adicionales para ejecutar el programa tras la inclusión del mecanismo de detección de errores, o ii) que, al insertar el mecanismo de detección de errores como parte inherente a la ejecución de instrucciones, se inserte sobre una ruta temporalmente crítica, obligando a reducir la frecuencia del reloj del sistema.

En resumen, las características ideales de un mecanismo de detección de errores (amén de sus parámetros como tal mecanismo, una máxima cobertura de detección con una mínima latencia) serían las siguientes:

- No modificar el juego de instrucciones original del procesador que se va a monitorizar
- No modificar la arquitectura del procesador que se va a monitorizar
- No requerir lógica adicional, ni memoria
- No influir en el tiempo de ejecución, de forma que las prestaciones resultantes sean las mismas antes y después de su inclusión.

Adicionalmente, otras características deseables son: i) que no dependa para su implantación de características específicas del procesador que va monitorizar y ii) que mantenga la máxima compatibilidad con el código binario existente, aún a costa de su objetivo primigenio, la detección de errores.

13.2. Aportaciones

Para realizar el trabajo que aquí se presenta se han analizado las diferentes propuestas para la inclusión de mecanismos de detección de errores del control de flujo de ejecución de un procesador, tanto software como hardware.

La aportación más novedosa (y núcleo central de este trabajo) ha consistido en la propuesta de una nueva técnica de empotrado de firmas derivadas en el espacio de instrucciones del procesador a monitorizar.

Esta técnica, denominada ISIS (acrónimo de *Interleaved Signatures Instruction Stream*), no depende de ninguna característica específica de la arquitectura o juego de instrucciones del procesador a monitorizar, lo que permite su utilización sobre cualquier arquitectura.

Las firmas se *intercalan* (de ahí el nombre de esta técnica) entre los bloques básicos de ejecución secuencial del programa original. Para la verificación de las instrucciones ejecutadas por el procesador principal no se requiere ninguna modificación al juego de instrucciones del procesador, lo que permite una compatibilidad binaria total con el software pre-existente. Como diferencia fundamental en la forma de realizar el empotrado de las firmas respecto de propuestas previas, y que resulta en una apreciable mejora de las prestaciones del sistema, estas firmas pasan completamente inadvertidas para el procesador principal gracias al cambio introducido en la *semántica* de las instrucciones de salto condicional.

El procesador principal, tras la ejecución de una instrucción actualiza el contador de programa para ejecutar la instrucción almacenada a continuación. El cambio semántico al que se alude anteriormente consiste en que el procesador, si durante la ejecución de una instrucción de salto condicional determina que la condición del salto no se cumple (y que por tanto la ejecución debe seguir el orden secuencial usual), actualiza el contador de programa de forma que "esquiva" una posición, evitando la búsqueda y ejecución de una palabra de memoria; el hueco así generado es aprovechado para insertar la firma del bloque.

Para permitir que el procesador de guardia tenga acceso a dicha palabra de forma simultánea al acceso a instrucciones del procesador principal, la memoria caché de instrucciones debe disponer de dos puertos de acceso, uno para el procesador principal y otro para el de guardia.

Tampoco se requiere que el programador realice ningún cambio al código fuente de los programas para utilizar eficazmente este mecanismo de detección de errores. En la implementación práctica de esta técnica de inserción de firmas sobre un procesador RISC de la familia MIPS se han modificado las herramientas software de desarrollo para conseguir que el sistema incorpore las firmas de forma automática con el simple añadido de una opción de compilación adicional.

Además de proponer una técnica original para el empotrado de firmas, y como segunda aportación de esta tesis, dentro de los mecanismos de verificación de la integridad estructural del programa ejecutado por el procesador principal se ha añadido una propuesta que permite verificar un salto cuando éste se produce a múltiples destinos (todos conocidos en el momento del enlazado) de una forma sencilla y elegante.

La verificación del salto cuando tiene múltiples destinos se consigue realizando la comprobación desde el bloque alcanzado. Esta verificación, denominada SAC (*Source Address Checking*) se consigue almacenando en cada uno de los posibles destinos un checksum del desplazamiento del salto recién ejecutado. Es la primera vez que se propone una fórmula de verificación *tras el salto* en la literatura de la materia. En las propuestas previas, cuando existen múltiples destinos y se desea verificar el salto, esta verificación se consigue mediante i) el uso de firmas de ajuste entre el camino más frecuente y los demás destinos posibles, o ii) utilizando algún tipo de tabla en el bloque origen conteniendo todos los posibles destinos.

La originalidad de la verificación SAC es que permite utilizar un solo campo (de cada uno de los nodos destino implicados) para realizar, tras el salto, una única comprobación para verificar que el salto se ha producido desde el bloque correcto.

El desarrollo práctico de las propuestas anteriores ha dado lugar a la tercera aportación de esta tesis: un modelo sintetizable del sistema desarrollado en el lenguaje de descripción de hardware VHDL, que junto al procesador principal de arquitectura MIPS incluye un procesador de guardia, memoria caché de instrucciones, bus intra-chip AMBA, interfaz con memoria externa, y todo lo necesario para su utilización en un sistema real sobre un dispositivo lógico

programable.

A este sistema se le ha denominado HORUS y es la aportación práctica más importante del trabajo descrito en esta memoria.

La estructura de HORUS permite también la ejecución de procesos que no llevan asociadas firmas para la verificación de la integridad de las instrucciones que se ejecutan en el procesador, pudiendo activar y desactivar la detección de errores en función de la tarea en ejecución, lo que permite una migración incremental hacia la inclusión de firmas en las diferentes tareas de un sistema y una total compatibilidad con el software ya existente.

El sistema HORUS permite la utilización usual de interrupciones y excepciones en el sistema, que en propuestas de otros procesadores de guardia no era posible. Hay que tener en cuenta, sin embargo, que la latencia de detección se ve negativamente influenciada si, durante la ejecución de un bloque de instrucciones con un error y antes de que la ejecución de éste termine, el procesador pasa a ejecutar el manejador de una interrupción o excepción, ya que la detección no se produce hasta que el procesador principal no ejecuta la última instrucción del mencionado bloque.

Una interesante característica de la estructura interna de HORUS es que la pérdida de prestaciones no supera el 6 % del tiempo de ejecución en ninguna de las pruebas llevadas a cabo, quedando en muchos casos por debajo del 2 %. Lamentablemente, comparte con las demás propuestas de procesadores de guardia unos requerimientos de memoria que pueden llegar a incrementar las necesidades de espacio de memoria de los programas hasta el 30 %.

Con HORUS se ha querido demostrar la factibilidad práctica de las propuestas lanzadas con ISIS, de manera que el trabajo tiene una inmediata aplicación. También viene este desarrollo a solventar las dudas (que por otra parte surgen de forma natural) sobre la posibilidad de incorporar de una forma *completamente automática* las firmas sobre un código fuente sin que el programador tenga que hacer modificación alguna. Estas dudas quedan solventadas con la cuarta aportación de esta tesis, que consiste en la modificación de aplicaciones de sobra conocidas para que la generación e inserción automática de las firmas de referencia sea una realidad: el conjunto de utilidades conocido como *binutils* que acompaña al compilador `gcc` de GNU, fundamentalmente el ensamblador `gas`, el enlazador/cargador `ld` y la librería `bfd` que permite gestionar el formato del código objeto (en nuestro caso el formato *elf* o *Executable and Linkable Format*). Se ha elegido este entorno de desarrollo por varias razones, entre las que cabe destacar:

1. Que, al ser un desarrollo de código abierto, cualquier modificación puede ser incorporada sin mayores problemas.

2. Que, dada la propia estructura de funcionamiento del proceso de compilación/enlazado, los cambios realizados al ensamblador, enlazador y librería base son automáticamente aplicados a cualquier lenguaje de alto nivel soportado por el procesador `gcc`.

3. Que las herramientas de GNU gozan de un amplio reconocimiento por su versatilidad y aplicación en el desarrollo de aplicaciones de propósito general y en el ámbito industrial, siendo utilizadas todos los días para el desarrollo de aplicaciones reales. No se trata pues de una herramienta meramente académica, lo que despeja las posibles dudas de su posible utilización en sistemas reales.

Como aportación final, conjuntamente al desarrollo de HORUS se han creado una serie de de rutinas software para la inyección de fallos en el modelo VHDL del sistema, a fin de verificar la funcionalidad del procesador de guardia. Este conjunto de utilidades, escrito completamente en el lenguaje de *scripts* `tcl` se ha denominado FIASCO (acrónimo de *Fault Injection Aid Software COmponents*).

13.3. Conclusiones

Del conjunto de experimentos de inyección de fallos llevado a cabo para verificar la funcionalidad del sistema sólo podemos concluir que la cantidad de fallos que sería necesario inyectar para caracterizar los mecanismos de detección de errores es enorme.

A pesar de haber inyectado algunos miles de fallos, generados aleatoriamente y repartidos uniformemente por todo el sistema, todos aquellos que han producido una avería en el sistema de las que el procesador de guardia está preparado para detectar han sido detectados por algún mecanismo, bien del procesador de guardia (integridad de la longitud del bloque, de las instrucciones ejecutadas, del salto realizado) o bien del sistema original (accesos a posiciones de memoria ilegales detectadas por la MMU del procesador, por ejemplo).

Como no es posible desarrollar un mecanismo de detección con una cobertura del 100 %, y puesto que es fácil imaginar algún caso que el procesador de

guardia no puede detectar, se deduce sencillamente que para que en una campaña de inyección de fallos generada de forma aleatoria y uniforme sobre todo el sistema aparezcan esos casos críticos que hacen bajar la cobertura del más que improbable 100 % es necesario inyectar varias decenas o incluso centenares de miles de fallos en el sistema.

Por último, del análisis de prestaciones y consumo de memoria del sistema resultante podemos concluir que:

1. Una de las causas de la pérdida de prestaciones del procesador principal reside en la interferencia que sobre el funcionamiento de la memoria caché de instrucciones produce el acceso a posiciones de memoria del procesador de guardia para la recogida de las firmas.

   Teniendo en cuenta que el procesador de guardia monitoriza las instrucciones que entran al pipeline del procesador principal no sólo para verificar si ha habido errores sino también para determinar dónde está la firma que debe utilizar como referencia, que dicha firma es requerida por el procesador de guardia en cuanto el procesador principal comienza la ejecución de un bloque secuencial básico, y que además la susodicha firma se almacena en la posición de memoria inmediatamente anterior a la primera instrucción del mencionado bloque, podemos deducir que las propias características que hacen que una memoria caché tenga un buen rendimiento (localidad espacial y temporal) ayudan a que la interferencia del procesador de guardia sea pequeña.

   Expresado en pocas palabras, la interferencia del procesador de guardia sobre el funcionamiento de la caché de instrucciones es pequeña porque las posiciones requeridas por éste coinciden en el espacio y en el tiempo con las instrucciones que está ejecutando el procesador principal.

2. Otra de las causas de la pérdida de prestaciones del procesador principal reside en el hecho de que, a pesar de que este último no ejecuta ni requiere las firmas de cada bloque, lo cierto es que en algunos casos es necesario insertar nuevas instrucciones al programa del procesador principal que originalmente no aparecían.

   Estas instrucciones han de ser insertadas cuando, por ejemplo, el número de instrucciones del bloque secuencial supera la cantidad máxima prevista en el campo de longitud de la firma asociada. En este caso, el sistema de generación de firmas inserta una instrucción de salto para forzar la división en varios bloques secuenciales de menor tamaño. Estas instrucciones de salto sí son recuperadas de la memoria y ejecutadas

por el procesador principal, y puesto que no existían en el programa original, el tiempo dedicado a su ejecución ha de ser considerado como tiempo perdido por el procesador principal.

Esta penalización en las prestaciones puede aliviarse sin más que aumentar la longitud máxima permitida a un bloque secuencial, fijada en 16 instrucciones en la propuesta de ISIS. Debe tenerse en cuenta, sin embargo, las implicaciones que este aumento tendrá sobre el sistema resultante:

a) Al aumentar el campo de longitud en la firma de un bloque es necesario reducir alguno de los otros campos dedicados a la detección de errores. Será necesario evaluar, pues, cómo esa reducción afecta a la cobertura del mecanismo afectado.

b) El hecho de permitir un bloque secuencial de mayor longitud implica que, en el peor de los casos, transcurre más tiempo (el tiempo dedicado a ejecutar más instrucciones) desde la ocurrencia de un error hasta la detección; es decir, se está afectando negativamente a la *latencia de detección*.

c) También se está afectando negativamente a la probabilidad de detección (la *cobertura*) de un error sobre el código binario de las instrucciones, como sería por ejemplo la transformación de una instrucción de suma en una de resta al corromperse el código de operación. Esto es así porque la probabilidad de detección del fallo, manteniendo fija la longitud del código utilizado como respaldo, disminuye al incrementar la cantidad de instrucciones involucradas; el efecto en este caso es, sin embargo, de mucho menor calado dadas las excelentes características de detección de los códigos de redundancia cíclica empleados.

3. Las necesidades de espacio de memoria para utilizar las propuestas de ISIS están muy ligadas a la estructura del programa que se pretende verificar.

   Por un lado, la existencia de bloques secuenciales demasiado largos para el procesador de guardia forzará, durante el ensamblado, la inserción de instrucciones de salto que dividan dichos bloques en otros de menor tamaño.

   Por otro lado, la existencia de muchos bloques de pequeño tamaño supone un incremento relativo muy alto del consumo de memoria, dado que cada bloque (de la longitud que sea) requiere de una firma (que

equivale en espacio de memoria a una instrucción del procesador principal). Evidentemente el incremento relativo de memoria para insertar una firma de un bloque de 4 instrucciones es del 25 %, pero sólo del 12.5 % si dicho bloque tiene 8 instrucciones.

A menos que se cambie la formulación del compilador en su traducción del código de alto nivel para que tienda a generar bloques de tamaño intermedio (si ello es posible, claro), la forma de reducir esta sobrecarga de memoria estará en:

a) Reducir el tamaño de la firma de un bloque secuencial, siempre y cuando esta reducción permita reducir el consumo de posiciones de memoria. No es el caso de la arquitectura MIPS, dado que toda instrucción es de 4 bytes y reducir la firma a la mitad no permitiría aprovechar los dos bytes de hueco resultante (las instrucciones deben mantenerse alineadas en direcciones múltiplo de 4). Sin embargo, si sería en principio posible en otras arquitecturas, como por ejemplo la MIPS16, que utiliza instrucciones de 2 bytes alineadas en direcciones pares.

b) Eliminar la firma de un bloque secuencial como una información separada del programa principal, de manera que dicha información se "integre", por ejemplo, en los huecos no utilizados de las instrucciones del procesador principal.

13.4. Publicaciones directamente relacionadas con el trabajo de tesis

La propuesta de empotrado de firmas y el conjunto de mecanismos de detección de ISIS junto con la descripción de la arquitectura del procesador HORUS que las implementa fueron publicados en

- F. Rodríguez, J.C. Campelo, J.J. Serrano. *A Watchdog Processor Architecture with Minimal Performance Overhead.*
  Proc. SAFECOMP'02, Lecture Notes in Computer Science, vol. 2434, pp. 261-272, Catania, Italia, 2002

- F. Rodríguez, J.C. Campelo, J.J. Serrano. *The HORUS Processor.*
  Proc. XVII Conference on Design of Circuits and Integrated Systems (DCIS'2002), Santander, España, 2002

Los resultados sobre el análisis de prestaciones y el consumo de memoria se publicaron en los artículos que a continuación se enumeran; en el último trabajo se apuntan ya posibles soluciones para reducir el consumo de memoria sin afectar negativamente las prestaciones.

- F. Rodríguez, J.C. Campelo, J.J. Serrano. *Delivering Error Detection Capabilities into a Field Programmable Device: The HORUS Processor Case Study*. Proc. 2002 IEEE International Conference on Field-Programmable Technology (FPT'2002), Hong-Kong, China, 2002
- F. Rodríguez, J.C. Campelo, J.J. Serrano. *A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream*. Proc. 17th IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'02), Vancouver, Canadá, 2002
- F. Rodríguez, J.C. Campelo, J.J. Serrano. *Improving the Interleaved Signature Instruction Stream Technique*. Proc. WSEAS Intl. Conference (ICAI'2002), Santa Cruz de Tenerife, España, 2002
- F. Rodríguez, J.C. Campelo, J.J. Serrano. *Improving the Interleaved Signature Instruction Stream Technique*. Proc. IEEE C. Conference on Electrical and Computer Engineering (CCECE'2003), Montreal, Canadá, 2003

En el siguiente trabajo se puede encontrar la descripción de las modificaciones a la suite *binutils* para la generación automática de firmas y sobre los experimentos realizados para obtener la cobertura de detección de los mecanismos de detección de errores basados en la dirección de salto.

- F. Rodríguez, J.J. Serrano. *Control Flow Error Checking with ISIS*. Proc. Intl. Conference on Embedded Systems and Software (ICESS'2005), Lecture Notes in Computer Science, vol. 3820, pp. 659-670, Xi'an, China, 2005

Finalmente, los trabajos publicados sobre la herramienta FIASCO y la inyección de fallos en el modelo VHDL del sistema HORUS utilizando un sistema distribuido para la reducción del tiempo del experimento fueron

- F. Rodríguez, J.C. Campelo, J.J. Serrano. *Reducing the VHDL-Based Fault Injection Simulation Time in a Distributed Environment*. Informal Digest Proc. 7th IEEE European Test Workshop (ETW'02), Corfú, Grecia, 2002

- F. Rodríguez, J.C. Campelo, J.J. Serrano. *A Distributed Simulation Environment for Fault Injection Analysis on SoC Models.* Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Intl. Workshop (DDECS'2002), Brno, República Checa, 2002

13.5. Trabajo futuro

Como trabajo futuro se plantean las siguientes líneas de actuación:

- Una primera línea de trabajo viene determinada por la caracterización del procesador de guardia implementado en el sistema HORUS, para determinar de forma experimental la cobertura de detección de errores y su latencia de detección.

  No se trataría de determinar estos parámetros sin más, sino de realizar un estudio más amplio para determinar la posibilidad de reducir los bits requeridos por la firma de los nodos (manteniendo una alta cobertura de detección, evidentemente) a fin de poder dotar de más bits a los campos utilizados en la verificación de los saltos.

  Se trata de encontrar un balance entre los bits dedicados a cada tipo de verificación para conseguir una alta cobertura de detección en todos ellos, sin perjudicar de forma significativa a ninguno.

- Otra vía natural de continuación de este trabajo es la incorporación al sistema de mecanismos de recuperación de errores. En el sistema actual, cuando se detecta un error el procesador de guardia activa una línea de salida de error. Esta línea se puede conectar a una señal interna para generar una excepción software en el procesador, o a una señal externa (para forzar la reinicialización del sistema, por ejemplo). Sin embargo, no se ha ido más allá en el tratamiento que el sistema debe dar a la detección de dicho error.

- Del estudio llevado a cabo sobre las diferentes técnicas de inserción de firmas se puede concluir que aún quedan algunos casos en las estructuras de ejecución que no pueden ser cubiertas por ningún mecanismo de detección.

  Es cierto que con la propuesta del trabajo que aquí se presenta se han añadido a la lista de situaciones detectables los saltos con múltiples

destinos (que anterioremente no era posible verificar) mediante la inclusión de un nuevo tipo de verificación en el que, una vez alcanzado el destino del salto, se verifica si el origen del mismo es correcto. Pero estos saltos con destino múltiple son verificables sólo en el caso de que, para cada uno de los destinos posibles, el origen sea único.

Sin embargo, cuando un bloque es alcanzado desde dos o más bloques origen con un salto a múltiples destinos, dicho bloque destino no puede incluir la verificación del nodo origen (pues éste debe ser único). Es necesario, por tanto, hacer un análisis de porqué aparecen este tipo de estructuras y de cómo incorporar algún tipo de mecanismo que permita verificar el salto realizado por el procesador, lo que permitiría ampliar el conjunto de saltos verificables.

En este sentido, una línea que parece prometedora es la réplica de estos bloques o la generación automática de bloques *puente*.

- Finalmente, cabe resaltar que sería interesante disponer de un mecanismo de detección de errores que no tuviera unos requisistos de memoria tan exigentes. Para ello ya se ha planteado en alguno de los artículos publicados mencionados en la sección anterior la posibilidad de empotrar las firmas de los bloques en los campos no utilizados de las instrucciones. Para aquellos bloques que no dispongan de huecos suficientes en las instrucciones originales habría que añadir instrucciones de no operación en los que insertar los bits restantes.

  Es necesario evaluar en este caso el incremento de complejidad del procesador de guardia, pues deberá ahora filtrar las instrucciones del procesador principal para extraer de ellas las firmas de los bloques, y si este incremento de complejidad se ve compensado con la simplificación de la memoria caché de instrucciones, que sólo requeriría un puerto de acceso (para el procesador principal).

  También será necesario hacer un estudio del juego de instrucciones de las arquitecturas más relevantes para determinar si existen en ellos huecos suficientes como para que esta modificación a la técnica de inserción de firmas tenga aplicación práctica.

Bibliografía

[1] María Eulalia Fuentes i Pujol. *Documentación científica e información: Metodología del trabajo intelectual y científico.* Escuela Superior de Relaciones Públicas: Promociones y Publicaciones Universitarias SP, 1992. Barcelona, España.

[2] A. Avizienis. Building dependable systems: How to keep up with complexity. In *Proceedings of the 25th Fault Tolerant Computing Symposium (FTCS-25)*, pages 4–14, 1995. Pasadena, California.

[3] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the 19th Fault Tolerant Computing Symposium (FTCS-19)*, pages 340–347, 1989. Chicago, Illinois.

[4] E. W. Czeck and D. P. Siewieorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 20th Fault Tolerant Computing Symposium (FTCS-20)*, pages 236–243, 1990. NewCastle Upon Tyne, U.K.

[5] J. Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error detection. In *Proceedings of the 27th Fault Tolerant Computing Symposium (FTCS-27)*, pages 42–46, 1997. Seattle, Washington.

[6] J. Ohlsson, M Rimén, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In *Proceedings of the 22th Fault Tolerant Computing Symposium (FTCS-22)*, pages 316–325, 1992. Boston, Massachusetts.

[7] D. P. Siewiorek. Niche sucesses to ubiquitous invisibility: Fault-tolerant computing past, present, and future. In *Proceedings of the 25th Fault To-*

lerant Computing Symposium (FTCS-25), pages 26–33, 1995. Pasadena, California.

[8] R. K. Iyer N. Nakka, Z. Kalbarczyk and J. Xu. An architectural framework for providing reliability and security support. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN-2004)*, pages 585–594, 2004. Florence, Italy.

[9] J. Gaisler. Concurrent error-detection and modular fault-tolerance in an 32-bit processing core for embedded space flight applications. In *Proceedings of the 24th Fault Tolerant Computing Symposium (FTCS-24)*, pages 128–130, 1994. Austin, Texas.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan-Kauffmann Publisher Inc., second edition, 1996.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control flow checking by software signatures. *IEEE Transactions on Reliability Special Section on Fault Tolerant VLSI Systems*, 51(2), March 2002.